Palo Alto Research Centers

# Design and Implementation of a Relationship-Entity-Datum Data Model

R. G. G. Cattell

XEROX

# Design and Implementation of a Relationship-Entity-Datum Data Model

R. G. G. Cattell

CSL-83-4 May 1983 [P83-00004]

© Copyright Xerox Corporation 1983. All rights reserved.

**Abstract:** The Model level of the Cypress Database Management System, built upon the earlier Cedar DBMS, provides data description and access capabilities at a higher level of abstraction than the existing system and other conventional DBMS's. In this report we describe the design of the Cypress data model and discuss issues in the efficient implementation of such a model. Cypress incorporates features motivated by experience with local database applications. It may be viewed as an integration of a number of existing data models; we present the criteria that led to this choice. The Cypress primitives include simple data values such as strings or integers, entities representing real or abstract objects, and relationships among entities and/or simple data values. We also provide mechanisms for a hierarchy of types, relational keys, and segmentation of databases into independent files. Cypress allows a conventional relational query language. We argue that our extensions to simpler data models allow a more powerful and efficient implementation, and we describe the optimizations Cypress performs. We also discuss some preliminary experience with user tools and applications developed in conjunction with Cypress.

**CR categories:** H.2.1, H.2.2, H.4.1

**Key words and phrases:** Cedar, Cypress, database systems, semantic model, data model

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

# TABLE OF CONTENTS


1. Introduction 1
   - 1.1 Data modelling
   - 1.2 Cypress and Cedar
   - 1.3 Design criteria and motivation

2. Cypress data model concepts 5
   - 2.1 Data independence
   - 2.2 Basic primitives
   - 2.3 Names and keys
   - 2.4 Basic operations
   - 2.5 Aggregate operations
   - 2.6 Convenience operations
   - 2.7 Normalization
   - 2.8 Segments
   - 2.9 Augments
   - 2.10 Views
   - 2.11 Summary

3. Model level interface 27
   - 3.1 Types
   - 3.2 Transactions and segments
   - 3.3 Data schema definition
   - 3.4 Basic operations
   - 3.5 Query operations
   - 3.6 System domains and relations
   - 3.7 Errors

4. Application example 47
   - 4.1 A database application
   - 4.2 Schema design
   - 4.3 Example program

5. Data model design issues 57
   - 5.1 Relations and attributes
   - 5.2 Entities and domains
   - 5.3 Entities as relationships
   - 5.4 Relationships as entities
   - 5.5 Lists and sets
   - 5.6 Entity names
   - 5.7 Keys, normalization, and dependencies
   - 5.8 Generalization and type hierarchies
   - 5.9 Access primitives
   - 5.10 Views, segments, and augments
   - 5.11 Summary

6. Data model implementation issues 73
6.1 Storage level structures
6.2 Entities and relationships
6.3 Values
6.4 Caching the data schema
6.5 Links and colocation
6.6 Surrogate relations
6.7 Basic operations
6.8 Aggregate operations
6.9 Summary

7. Database environment and applications 87
7.1 Database environment
7.2 Database tools
7.3 Database applications
7.4 Summary

8. Status and conclusions 101
8.1 Summary
8.2 Some results
8.3 Status and plans
8.4 Summary

9. Annotated bibliography 105

10. Appendix and index 113

# 1. Introduction

## 1.1 Data modelling

A *data model* is a scheme for describing the types of data that may be stored in a database: how these data may be structured, and how they may be accessed. Any particular database consists of the data themselves plus a *data schema* that describes the types of data in terms of the data description primitives of the data model. The three data models most widely used in the past are the network, hierarchical, and relational models. A good summary of these models can be found in *Computing Surveys*, March 1976.

In recent years, data models with more sophisticated representation schemes have been proposed, often called *semantic data models*. Unfortunately, this more recent work is voluminous and difficult to understand: their terminology is mutually conflicting, the models are complex, and related work was done concurrently by a number of authors. Furthermore, almost none of the models were actually implemented. Related problems of data structure specification have also been addressed extensively in the programming language and artificial intelligence literature, as abstract data type mechanisms and knowledge representation languages. A survey of the literature is beyond the scope of this document. An annotated bibliography is included in the last section, however, and Tsichritzis & Lochovsky [1982] summarize much of the recent data modelling work.

The data model described herein is the result of analyzing the strengths and weakness of a number of proposed models, and integrates a variety of viewpoints. It includes only those features that are accepted in some form in a number of models, or proved particularly useful for our database applications. We will call our model the Cypress data model. It might alternately be called the Relationship-Entity-Datum model, after its three basic primitives.

The Cypress data model is described in Section 2. Sections 3 and 4 provide a description of the programmer's interface to our implementation, and an example of its use. Section 5 contrasts the Cypress data model to others in the literature, and describes how we arrived at this particular integration of data modelling ideas. Section 6 describes issues in the implementation of the Cypress data model; as none of the models we reference have actually been implemented, this is an important result of the work. To complete our description of the Cypress work, Section 7 covers experience with database access tools and applications built upon the system. Sections 8 and 9 provide a summary and annotated bibliography. An appendix of formal axioms for the model and an index of important terms are also included.

For the sake of clean exposition, Section 2 covers only the abstract model, not its implementation or

rationale. The reader interested only in the basic ideas may read about the data model in Section 2 and about the applications it enables in Section 7, without loss of continuity. A potential client of Cypress should read Sections 2, 3, and 4.

### 1.2 Cypress and Cedar

The motivation for design and implementation of a database system in the Computer Science Laboratory arose from the needs of anticipated and existing database applications running in the Cedar Programming Environment. The design of a data model was deferred to the second phase of the earlier Cedar Database Managment System (DBMS) project to allow a choice of model after some experience with our needs. With the second phase of development, the name of the Cedar DBMS was changed to Cypress, to distinguish it from the Cedar DBMS which it replaced, the Cedar Programming Environment in which it provides database facilities, and the Cedar Mesa Programming Language in which it is written.

The original Cedar DBMS, described by Brown, Cattell, and Suzuki [1980], implemented tuples whose elements are integers, strings, and references to other tuples. It provides a large virtual memory of tuples which can be moved or deleted safely, optional B-trees for indexing of tuples, and concurrent transaction-based access to data over a network. It was built in three levels: the Cache, Storage, and Tuple Levels. The latter (top) level provides primitive query mechanisms, and does little or no checking of the integrity of data type, form, uniqueness, or referents; the addition of the Model Level introduces types to the essentially typeless underlying system.

Initial clients of the Cedar DBMS included:

1. The CSL Notebook, a database of memoranda from laboratory members,
2. PDB, a personal database including notes, bibliographies, addresses, phone numbers, etc.,
3. A database of components of large Mesa (Mitchell et al [1976]) systems and their inter-dependencies,
4. A database of Mesa source programs decomposed to the level of procedures, types, and variables,
5. A database of data and events in an automated office system.

Most of these applications built their own data model on top of the existing system, to enable the data structures and access mechanisms they required. In some cases, building on top as opposed to

integrating the data model with the existing system (an option not available to them) led to difficulties with the performance or integrity of the total system. This is not surprising, even though the original system was designed to allow several data modelling choices at a future date; the choice of data model must influence the choices at all levels of the system, if adequate performance is to be obtained.

The addition of a data model to the database system makes possible the development of general-purpose tools. The more data semantics encoded in the database, the more the database system and its associated tools may provide without specific knowledge of an application's semantics. In order to print out data in a meaningful form, for example, a tool must know which data comprises names for objects. In order to abbreviate and/or check the type of input data, a tool must know what types of data may be related to what others in what ways. In order to efficiently represent potentially circular pointer structures linearly, a tool needs a specification of allowed data relationships. And so on.

The introduction of a data model also simplifies sharing a single database among multiple applications. A hierarchy of types allows different applications to have differing perspectives on the same objects. The logical integrity checks help protect the applications from one another. More sophisticated data modelling mechanisms allow applications different independent views of the data or different physical environments (files) for their data. Sharing a database between applications is important to avoid redundant, inconsistent representation of the same information, to mutually benefit from multiple data input sources, and, most importantly, to present one simple data access and manipulation mechanism to computer users. This contrasts, for instance, to three separate personnel databases (and access tools) for applications dealing with phone numbers, electronic mail, and laboratory bibliographies.

Finally, the introduction of the Model Level data model is important to future plans for the next level of the Cypress DBMS, the Query Level. At the Query Level, a database query language will be implemented to allow access and/or updates to individual data items or data aggregates, in a concise form amenable to optimization and decoupled operation in a database server on a computer network. The Model Level provides the functions to enumerate the database objects satisfying the queries parsed by the Query Level.

In summary, the development of the Model Level of the Cypress DBMS is a follow-on to the Cache, Storage, and Tuple Levels, a prerequisite of the Query Level, and motivated by:

1. the perceived needs of, or shortcomings for, future and existing database applications,

2. facilitation of shared databases for partial or complete integration of multiple applications,

3. the need for more data semantics to enable useful database utilities, e.g., a browser, and

4. the need for a query language for database access.

### 1.3 Design criteria and motivation

How do we choose a data model? Why should one data model be better than another? In most cases we can find a representational isomorphism between two models, such that we can automatically map between the two equivalent representation schemes. However, the models may still differ in the data access primitives they provide and the ease with which a user can understand and operate within the model. The arguments concerning ease of understanding are generally quite subjective, however, and will be avoided here except where these arguments are generally well-accepted in the literature.

The choices in the development of the Cypress Data Model were made on the basis of two criteria: *simplicity* and *utility*. Simplicity means ease of understanding and also representational parsimony, i.e., avoiding two or more mechanisms to represent the same semantics. Utility means the degree to which the data model avoids writing application-specific code for some function or integrity check; every feature need not be of use to *all* applications, though the feature should be of negligible cost to those that do not require it.

Simplicity and utility can of course be mutually conflicting; the result must be a balance incorporating the features deemed important for most applications, in the simplest form discovered. To make the presentation less confusing, we separate the discussion of the model itself in the sections to follow from the analysis of the simplicity and utility of its features in Section 5 and the discussion of its implementation in Section 6.

## 2. Cypress data model concepts

In this section, we give an informal description of the Cypress data model. The evaluation and justification of the model have been deferred to Section 5. A more formal description of the model has been deferred to a future paper (some axioms can be found in the appendix).

### 2.1 Data independence

We deal here with the *conceptual data model*, the logical primitives for data access and data type definition. This should be carefully distinguished from the *physical data model*, the mechanisms we are given for actual storage and access of data. The physical model in the Cypress implementation corresponds to the Storage level. The physical data model is hidden as much as possible from the database client to facilitate *data independence*, the guarantee that a user's program will continue to work (perhaps with a change in efficiency) even though the physical data representation is redesigned.

For any particular database using the given conceptual and physical *models*, the actual specifications of this database using the primitives the models provide are termed the *conceptual data schema* and *physical data schema*. Note that a mapping must be provided between the conceptual and physical levels, either automatically or with further instruction from the client; we will do some of both. The logical to physical mapping is intimately associated with the performance of the database system as viewed by the user performing operations at the conceptual level. Performance is generally not a criteria for choosing between conceptual data models, unless there is no known way to map the differing conceptual views into the same or equally efficient implementations.

### 2.2 Basic primitives

Three basic primitives are defined in the model: an *entity*, *datum*, and *relationship*.

An entity represents an abstract or concrete object in the world: a person, an organization, a document, a product, an event. In programming languages and knowledge representation entities have variously been referred to as atoms, symbols, and nodes. A datum, unlike an entity, represents literal information such as times, weights, part names, or phone numbers. Character strings and integers are possible datum types.

It is a *policy* decision whether something is represented as an entity or merely a datum: e.g., an employee's spouse may be represented in a database system as a datum (the spouse's name), or the spouse may be an entity in itself. The database system provides a higher level of logical integrity checking for entities than for datum values, as we will see later: unique entity identifiers, checks on

entity types, and removal of dependent data upon entity deletion. We shall discuss the entity/datum choice further in Section 4.2.

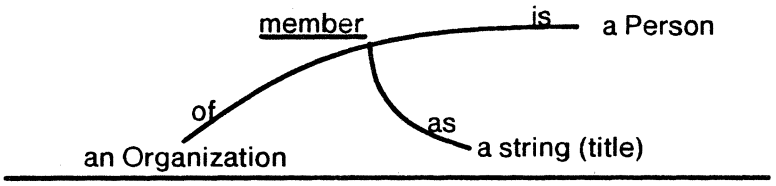
We will use the term *value* to refer to something that can be either a datum or an entity. In many programming languages, there is no reason to distinguish entity values from datum values. Indeed, most of the Cypress operations deal with any kind of value, and some make it transparent to the caller whether an entity or datum value is involved. The transparent case makes Relational operations possible in our model, as we will see in Section 2.5.

A *relationship* is a tuple whose elements are [entity or datum] values. We refer to the elements (fields) of relationships by name instead of position. These names for positions are called *attributes*.

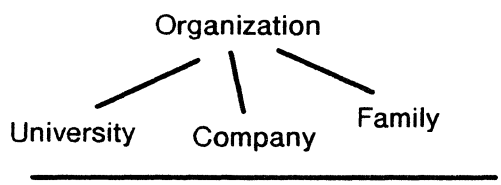
Note that we have separated the representatives of unique objects (entities) from the representation of information *about* objects (relationships), unlike some object-oriented programming languages and data models. Therefore an entity is *not* an "object" (or "record") in the programming language sense, although entities *are* representatives of real-world objects. We discuss the advantages of the entity-relationship distinction along with other data model design issues in Section 5.

We also define entity types, datum types, and relationship types. These are called *domains*, *datatypes*, and *relations*, respectively. We make use of these three types through one fundamental type constraint: every relationship in a relation has the same attributes, and the values associated with each attribute must be from a pre-specified domain or datatype. One might think of a relation as a "record type" in a programming language, although relations permit more powerful operations than record types.

As an example, consider a *member* relation that specifies that a given person is a member of a given organization with a given job title, as in the following figure. The person and organization might be entities, while the title might be a string datum. We relax the fundamental type constraint somewhat in allowing a *lattice* of types of domains: a particular value may then belong to the pre-specified domain *or* one of its sub-domains. For example, one could be a member of a University, a Company, or any other type of Organization one chooses to define. Other relations, e.g. an "offers-course" relation, might apply only to a University.

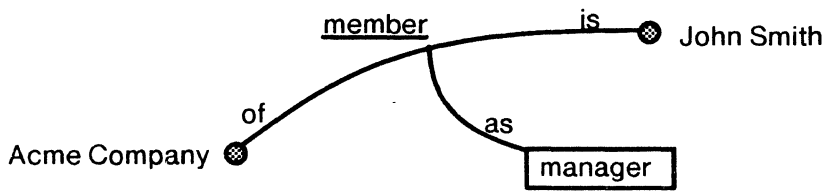



The member relation, its attributes (of, as, is) and their types




Some subdomains of Organization

Relationships represent facts about the world. A relation (a set of relationships) represents a *kind* of relationship in which entities and values can participate. We will schematically represent that John Smith is a member of Acme company with title "manager" by drawing lines for the relationship (labelled with the relation and its attributes), a circle for each entity, and a box for the datum:




One should normally think of relations as types, not sets. A relation can be defined as the set of all relationships of its type, however, and thus can be treated as a relation in the normal mathematical sense. Note that we differ from mathematical convention in another minor point: the use of attributes to refer to tuple elements by name instead of position. Reference by position is therefore not necessary and is in fact not permitted. We can often omit attribute names without ambiguity since the types of participating entities imply their role in a relationship. However, they are necessary in the general case; e.g., a *boss* relation between a *person* and a *person* requires the directionality to define its semantics.

We can summarize the six basic primitives of the data model in tabular form. Familiarity with these six terms is essential to understanding the remainder of this document:

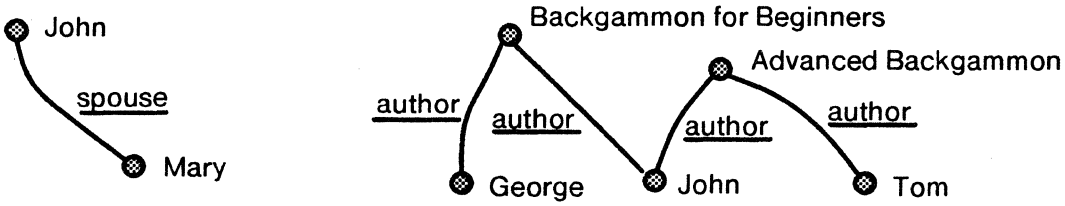
| Type | Instance | Instance Represents |
|---|---|---|
| Domain | Entity | physical or abstract object |
| Datatype | Datum | numerical measurement or symbolic tag |
| Relation | Relationship | logical correspondence between one or more objects and values |

Our terminology might be more consistent if we called a domain an "entity type," and a relation a "relationship type." Instead we have compromised on the terms most widely used in the literature for all six of the basic concepts. *The reader will find the remainder of this document much more understandable if these terms are commited to memory before proceeding.*

### 2.3 Names and keys

The data model also provides primitives to specify uniqueness of relationships and entities. For relationships, this is done with *keys*, and for entities, with *names*. A *key* for a relation is an attribute or set of attributes whose value uniquely identifies a relationship in the relation. No two relationships may have the same values for a key attribute or attribute set (a relation may have more than one key, in which case relationships must have unique values for all keys). A *name* acts similarly for a domain. The name of an entity must uniquely specify it in its domain.

Consider a *spouse* relation, a binary relation between a person and a person. Both of its attributes are keys, because each person may have only one spouse (if we choose to define spouse this way!). For an *author* relation, neither the person nor the document are keys, since a person may author more than one document and a document may have more than one author. The person and document attributes together would comprise a key, however: there should be only one tuple for a particular person-document pair.




We have labelled entities with names in the figures. Names are most useful when they are human-sensible tags for the entities, e.g. the title for a document, or the name of a person. However, their primary function is as a unique entity identifier, so non-unique human-sensible names must be represented as relations. If entities of a domain have more than one unique identifier, e.g. social security numbers and employee numbers, then one identifier must be chosen as the domain's entity names and the other represented as a relation connecting the entities with the unique alternate identifier (a key of that relation).

We require that every entity have a unique name, although the name may automatically be generated by the database system. Thus every entity may be uniquely identified by the pair:

[domain name, entity name].

Some authors in the database literature use the term entity to refer to a real-world object rather than its representation in the database system. When we use the term entity, we refer to the *internal entity*, the entity "handle" returned by database operations and stored in entity-valued variables, not the *external entity* that the internal entity represents or the *entity identifier* [domain,

name] that may be used to uniquely identify an internal entity. The three are interchangeable, however, since they must always be in one-to-one correspondence.

The reader may find it simple to think of entity-valued attributes of relationships as *pointers* to the entities, in fact bi-directional pointers, since the operations we provide allow access in either direction. This is a useful analogy. However, there is no constraint that the model be implemented with pointers, and the relationships of a relation could equally well be conceptualized as rows of a table whose columns are the attributes of the relation and whose entries for entity-valued attributes are the string names of the entities. For example, the *author* relationships in the previous figure could also be displayed in the form:

*Author:*

| Person | Book |
|---|---|
| George | Backgammon for Beginners |
| John | Backgammon for Beginners |
| John | Advanced Backgammon |
| Tom | Advanced Backgammon |

Thus our introduction of entities to the Relational model does *not* entail a different representation than, say, a Network model might imply, but simply additional integrity checks on entity names and types, and new operations upon entities. This compatibility with the Relational data model is important, as it allows the application of the powerful Relational calculus or algebra as a query language. We return to query languages in Section 2.5.

Note that the only information about an entity associated *directly* with the entity is the name; this contrasts with most other data models. A person's age or spouse, for example, would be represented in the Cypress data model via age or spouse relations. Thus the relationships in a database are the information-carrying elements: entities do not have attributes. However the model provide an abbreviation, *properties*, to access information such as age or spouse in a single operation. We will discuss properties later. In addition, the physical data modelling algorithms can store these data directly with the person entity as a result of the relation key information (since a person can have only one age, a field can be allocated for that field in the stored object representing a person.)

### 2.4 Basic operations

The data model provides the capability to define and examine the data schema, and perform operations on entities, relationships, and aggregates of entities and relationships. In this section we discuss the basic operations on entities and relationships. In Section 2.5, we discuss the operations

on aggregate types, i.e. domains and relations. We defer to Section 2.6 the discussion of "convenience" operations built upon the basic and aggregate operations.

Four basic operations are defined for entities:

1. **DeclareEntity[domain, name]**: Returns a new or existing entity in a domain. An entity name must be specified.
2. **DestroyEntity[entity]**: Destroys an entity; this also destroys all relationships that refer to it.
3. **DomainOf[entity]**: Returns the domain of an entity (its type).
4. **NameOf[entity]**: Returns the string name of an entity.

Five basic operations are defined for relationships:

1. **DeclareRelationship[relation, list of attribute values]**: Returns a relationship with the given attribute values in the given relation.
2. **DestroyRelationship[relationship]**: Destroys a relationship.
3. **RelationOf[relationship]**: Returns a relationship's relation.
4. **GetF[relationship, attribute]**: Returns the value associated with the given attribute of the given relationship.
5. **SetF[relationship, attribute, value]**: Sets the value of the given relationship attribute.

The operations upon relationships recognize a specially-distinguished *undefined* value for an attribute. Unassigned attributes of a newly-created relationship have this value. A client of the data model may retrieve a value with **GetF** and test whether it equals the distinguished undefined value, and may set a previously defined value to be the distinguished undefined value with **SetF**.

Other "convenience" operations are built on top of the basic operations on entities and relationships: *properties* and *translucent attributes*. They are described in Section 2.6. Although these operations are not essential to the basis of the Cypress model, they do furnish a fundamentally different perspective on the model. They provide a mechanism to associate information directly with entities (instead of through relationships) and to write programs largely independent of attribute types.

The reader will also note that we have ignored issues of concurrent access and protection in the basic operations. We will see later that an underlying transaction, file, and protection is associated with the relation and domain handles used in the basic operations. This convenience allows us to treat concurrency, protection, and data location orthogonally.

## 2.5 Aggregate operations

There are two kinds of operations upon domains and relations, the *aggregate types* in our model: the definition of domains and relations, and queries on domains and relations. We first discuss their definitions.

*Schema definition*

As in other database models and a few programming languages, the Cypress model is self-representing: the data schema is stored and accessible as data. Thus application-independent tools can be written without coded-in knowledge of the types of data and their relationships.

Client-defined domains, relations, and attributes are represented by entities. These entities belong to special built-in domains, called the *system* domains:

the **Domain domain**, with one element (entity) per domain

the **Attribute domain**, with one element per attribute

the **Relation domain**, with one element per relation

There is also a predefined **Datatype domain**, with pre-defined elements **StringType**, **BoolType**, and **IntType**, called *built-in types*. An implementation may also allow client-defined datum types, but the implementation described in Section 3 currently does not.

There may be other system domains, depending upon the implementation of the Relationship-Entity-Datum model, for example to represent indices on relations. The Domain domain, Attribute domain, and all other domains are members of the Domain domain.

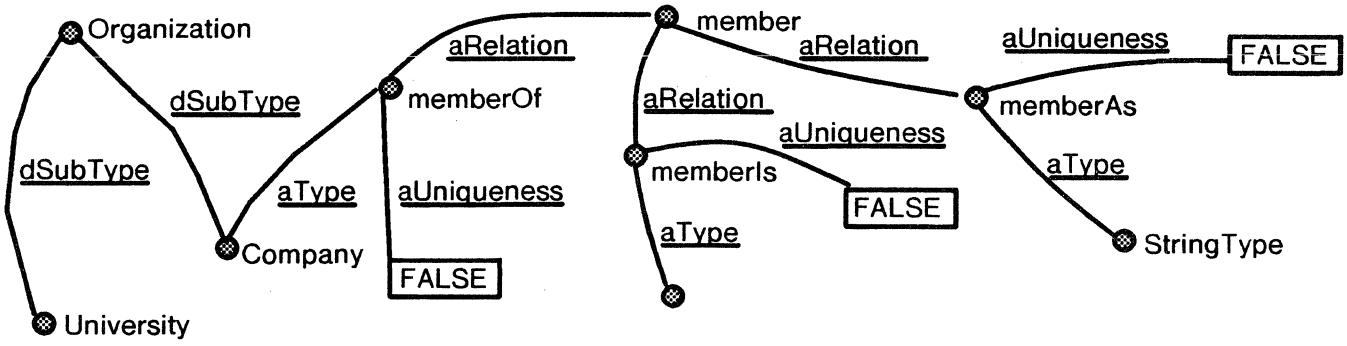
Information about domains, relations, and attributes are represented by *system relations* in which the system entities participate. The pre-defined **SubType** relation is a binary relation between domains and their subdomains. There are also predefined binary relations that map attributes to information about the attributes:

**aRelation**: maps an attribute entity to its relation entity.

**aType**: maps an attribute to its type entity (a domain or a built-in type)

**aUniqueness**: maps an attribute entity to {TRUE, FALSE}, depending whether it is part of a key of its relation. We are assuming only one key per relation, here; our implementation relaxes this assumption in the case of single-attribute keys.

The following diagram graphically illustrates a segment of a data schema describing the **member** relation and several domains. The left side of the figure shows two subdomains of **Organization**, (**Company** and **University**), and the right shows the types and uniqueness properties of the **member** relation's attributes **memberOf**, **memberIs**, and **memberAs**.




New domains, relations, and attributes are defined by creating entities and relationships in these pre-defined system domains and relations. However, an implementation of the model may choose to provide special operations to define the data schema, to simplify error checking. These operations are:

DeclareDomain[name], and for each subtype relationship:

DeclareSubType[superdomain, subdomain]

DeclareRelation[name], and for each attribute:

DeclareAttribute[name, relation, type, uniqueness].

*Queries*

The operation RelationSubset[relation, attribute value list] enumerates relationships in a relation satisfying specified equality constraints on entity-valued attributes and/or range constraints on datum-valued attributes. For example, RelationSubset might enumerate all of the relationships that reference a particular entity in one attribute and have an integer in the range 23 to 52 in another. Attributes of a relationship with an undefined value satisfy no range or equality constraints on the attribute.

The operation DomainSubset[domain, name range] enumerates entities in a domain. The enumeration may optionally be sorted by entity name, or restricted to a subset of the entities with a name in a given range.

More complex queries can be implemented in terms of DomainSubset and RelationSubset. A good implementation of the model might also provide a MultiRelationSubset operation to enumerate single queries spanning more than one relation. MultiRelationSubset operates upon a parsed representation of the query language, and produces the same kind of enumeration as RelationSubset.

We will not precisely define the query language for the Cypress data model (the Query level) in this report, and MultiRelationSubset has not yet been designed or implemented. A variety of query

languages (and MultiRelationSubset implementations) could be built on top of the operations we have defined thus far. For the purposes of this report, however, we will assume that our query language is the Relational algebra (Codd[1970]), providing the *join*, *projection*, and *selection* operators on relations. We choose the relational algebra due to its wide acceptance in the literature, and now commercially. Our choice does not preclude an entity-centric query language in future work, however. We and others have designed but have not implemented such languages.

To demonstrate that the relational algebra can be used with our system, we will define a mapping of the relational operations onto the Cypress model. Some augmentation of the relational algebra is necessary to include the operations our model provides upon domains. The important aspects of the mapping are:

1. The Cypress relations exist in our relational mapping essentially unchanged, except that some type constraints exist on attribute values; these constraints simply appear as exceptional conditions when violated, they do not effect the form of the relational operations.

2. For each domain in our schema, we define a unary relation in our mapping with the entity name as its only attribute. The sole purpose of the unary domain relations is to specify the existence of entities in the domain. For example, a Person relation would contain one tuple per person entity in the database. The unary domain relations allow DomainSubset to be performed in the relational algebra as a *select* operation on the domain.

3. Entity-valued attributes of relations appear to have the *name* of the corresponding entity stored in them. There are no "atomic" entity values, so the ChangeName operation is expensive in this model. All references to an entity in the database appear as the string name of the entity, and the system generates an error if an entity-valued attribute is initiaized to an entity which does not exist in the domain.

Our representation of the data schema is as before: schema entities and relationships in the domain domain, relation domain, attribute domain, subtype relation, and attribute relations.

The join, project, and select operations in the relational algebra can now be performed by MultiRelationSubset, or by the appropriate calls to RelationSubset. Note that we must typically declare new relations for the result and intermediate results of computing such queries. A select operation on a domain may be used to determine whether a particular entity exists, or to enumerate the entities of a domain. A select operation on a relation may be used to find relationships that reference a particular entity. Query operations on the schema can be used to determine what relations might reference an entity of a particular domain.

The unary relations representing domains are used only to determine the existence of entities. An

implementation could automatically create entities in these domains when an entity-valued attribute of a relationship is assigned a previously non-existent entity name. *Note that domain relations may in fact be ignored altogether by the user if this is done.* Domain relations are purely an extension to the Relational model. We chose not to automatically create entities in our mapping, as we would lose the logical integrity check the entities provide.

### 2.6 Convenience operations

Some more convenient specialized operations are built upon the basic operations described in the previous two sections. They implement what we call *properties* and *translucent attributes.* Although theoretically speaking these operations add no power to the model, they permit a significantly different perspective on the data access and so should be thought of as part of the model.

*Properties*

Properties allow the client to treat entities as if they, like relationships, had "attributes." They provide the convenience of treating attributes of relationships that reference an entity as if they were attributes (or *properties*) of the entity itself. The property operations are:

1. GetPList[entity, attribute$_1$, attribute$_2$]: Attribute$_1$ and attribute$_2$ must be from the same relation. Returns the values of attribute$_1$ for all relationships in the relation that reference the entity via attribute$_2$. Attribute$_2$ may be omitted, in which case it is assumed to be the only other entity-valued attribute of the relation.

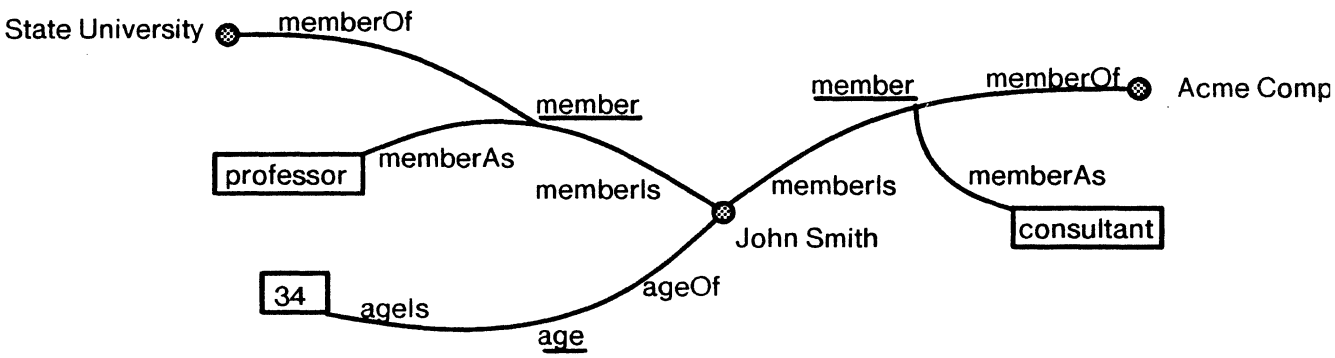
2. GetP[entity, attribute$_1$, attribute$_2$]: this is identical to GetPList except exactly one relationship must reference the entity via attribute$_2$; otherwise an error is generated. GetP always returns one value.

3. SetPList[entity, attribute$_1$, value list, attribute$_2$]: Attribute$_1$ and attribute$_2$ must be from the same relation. Destroys any existing relationships whose attribute$_2$ equals the entity, and creates new ones for each value in the list, with attribute$_1$ equal to the value, and attribute$_2$ equal to the entity. Attribute$_2$ can be defaulted as in GetPList.

4. SetP[entity, attribute$_1$, value, attribute$_2$]: this is identical to SetPList except it simply adds a new relationship referencing the entity instead of destroying any existing ones (unless attribute$_1$ is a key of its relation, in which case the existing one must be replaced).

Thus the property operations allow information specified through relationships to be treated as properties of the entity itself, in single operations. The property operations and the operations defined in earlier sections may be used interchangeably, as there is only one underlying representation of information: the relationships. As an example of the use of properties, consider

the following database:




The figure shows the entity **John Smith**, and three relationships in which he participates: an **age** relationship and two **member** relationships. The **member** relationships are ternary, the **age** relationship binary. On this database, the property operations work as follows:

> **GetPList[John Smith, memberOf]** would return the set **{Acme Company, State University}**.
>
> **GetP[John Smith, ageIs]** (or **GetPList**) would return **34**.
>
> **SetP[John Smith, memberOf, Foo Family]** would create a new **member** relationship specifying **John** to be a **member** of the **Foo Family**. **SetPList** would do the same, but would destroy the two existing **member** relationships referencing **John**. In either case, the **memberAs** attribute would be left undefined in the new relationship.
>
> **SetP[John Smith, ageIs, 35]**, where **ageOf** is a key of the age relation, would delete the relationship specifying **John**'s age to be **34**, and insert a new one specifying **John**'s age to be **35**. Note that **SetP** acted differently than on the **member** relation because **memberIs** is not a key.

Again, the property operations are simply a convenience, although they provide a different perspective on the data model by allowing an entity-based view of a database.

*Translucent attributes*

Some database application programs may not wish to be concerned with whether an attribute is entity-valued, string-valued, or integer-valued. They might prefer to have all values mapped to some common denominator, e.g. a string. An example would be a program that is simply displaying tuples on the screen.

Another class of applications would like to be independent of whether a particular attribute is represented as an entity or datum value. Consider the **member** relation in the previous figure. If

we choose to define an **Organization** domain, then the **memberOf** attribute is entity-valued; but instead we might choose to make the **memberOf** attribute be string-valued, merely giving the name of the organization without defining organizations as entities. This might be appropriate, for example, if we did not wish to invoke the type checking on uniqueness of names and the correctness of entity types. We would like to write programs that are independent of whether an attribute is string-valued or entity-valued (as in the Relational data model).

We introduce *translucent attributes* to avoid dependence on attribute types. Any attribute may be treated as a translucent attribute, by using the **GetFS** and **SetFS** operations to retrieve or assign its value.

**GetFS[relationship, attribute]** is identical to the **GetF** operation, except it returns a string regardless of the attribute's type. If the attribute is datum-valued, e.g. an integer or boolean, it is converted to a string equivalent. If the attribute is entity-valued, the *name* of the entity is returned.

**SetFS[relationship, attribute, value]** performs the inverse mapping. If the attribute is datum-valued, e.g. an integer or boolean, a string equivalent is accepted. If the attribute is entity-valued, the name of the entity is passed to **SetFS**. If an entity with the given name does not exist in the domain that is the attribute's type, then one is automatically created.

*Changing entity names*

Another convenience operation is provided on entities to change an entity's name: **ChangeName[entity, new name]**. This operation is semantically equivalent to destroying the given entity and creating a new one with the new name, participating in the same relationships that the old one did. See the description of ChangeName in Section 3.4 for precise semantics in our implementation, however.

## 2.7 Normalization

A common topic in the database literature is relational *normalization* (Codd [1970], Armstrong [1974], Hall et al[1976], Rissanen [1977], Beeri et al [1978], Fagin[1977, 1981], Biller[1979]). A relation is normalized by breaking it into two or more relations of lower order (fewer attributes) to eliminate undesirable dependencies between the attributes. For example, one could define a "publication" relation with three attributes:

*Publication:*

| Person | Book | Date |
|---|---|---|
| George | Backgammon for Beginners | 1978 |
| John | Backgammon for Beginners | 1978 |
| Mary | How to Play Chess | 1981 |
| Mary | How to Cheat at Chess | 1982 |

This relation represents the fact that John and George wrote a book together entitled "Backgammon for Beginners," published in 1978, and Mary wrote two books on the subject of chess, in 1981 and 1982. Alternatively, we could encode the same information in two relations, an author relation and a publication-date relation:

*Author:*

| Person | Book |
|---|---|
| George | Backgammon for Beginners |
| John | Backgammon for Beginners |
| Mary | How to Play Chess |
| Mary | How to Cheat at Chess |

*Publication-date:*

| Book | Date |
|---|---|
| Backgammon for Beginners | 1978 |
| How to Play Chess | 1981 |
| How to Cheat at Chess | 1982 |

Although the second two relations may seem more verbose than the first one, they are actually representationally better in some sense, because the publication dates of books are not represented redundantly. If one wants to change the publication date of "Backgrammon for Beginners" to 1979, for example, it need only be changed in one place in the publication-date relation but in two places in the publication relation. If the date were changed in only one place in the publication relation, the database would become inconsistent. This kind of behavior is called an *update anomaly*. The second two relations are said to be a normalized form (as it happens, third normal form) of the first relation, and thereby avoid this particular kind of update anomaly.

A variety of successively stricter criteria for normalization have been developed and studied, based on different kinds of real-world dependencies between attributes. Work on normalization will almost certainly continue through the forseeable future.

Relational normalization is not strictly part of the Cypress data model. However the model's operations (and the tools we will develop in the implementation) encourage what we will call *functionally irreducible* form, in which relations are of the smallest order that is naturally meaningful. This form is in some sense the most fully normalized canonical form that could be defined.

Functionally irreducible normal form cannot be defined simply syntactically, but rather requires resort to the semantics of relations. Specifically, the presence or absence of a relationship in a relation represents the truth of some *predicate* on the state of the represented world (Kent [1979]). For example, a "member" relationship represents the fact that a particular person is a member of a particular organization. (The *absence* of such a relationship may mean falsity of that predicate or lack of knowledge; we fortunately need not concern ourselves with this distinction here). A relation is in *irreducible form* if it is of the smallest order possible without introducing new artificial domain(s) not otherwise desired (all relations can be reduced to binary by introducing artificial domains). Biller [1979] provides a more precise definition of irreducible form. We will allow a slight weakening of irreducible form, functionally irreducible form, which permits combining two or more irreducible relations only when their semantics are mutually dependent (and therefore all present or absent in our world representation). For example, a *birthday* relation between a person, month, day, and year can be combined instead of using three relations. Another example would be an *address* relation between a person, street, city, and zip code. Combining an *age* and *phone* relation would not result in functionally irreducible form, however, as their semantics are not mutually dependent.

The functionally irreducible relations seen by the user are independent of the physical representation chosen by the system for efficiency, so we are concerned only with the logical data access. Note that in addition to avoiding update anomalies, functionally irreducible form provides a one-to-one correspondence between the relationships in the database and the atomic facts they represent, a canonical form that is in some sense more natural than any other form.

### 2.8 Segments

We would like a mechanism to divide up large databases, to provide different perspectives or subsets of the data to different users or application programs. In this section we discuss a mechanism to provide this separation: *segments*. A segment is a set of entities and relationships that a database client chooses to treat as one logical and physical part of a database.

In introducing segments, we will slightly change the definition of an entity, previously defined to be uniquely determined by its domain and name. We will treat entities with the same name and domain in different segments as different entities, although they may represent the same external entity. The unique identifier of an internal entity is now the triple

[segment, domain, name].

A consequence of this redefinition of entities is that relations and domains do not span segments, either. Application programs must maintain any desired correspondence between entities, domains, or relations with the same name in different segments. We will return to this later. In the next section, we will discuss a more powerful but more complex and expensive mechanism, *augments*, in which the database system itself maintains the correspondence.

We introduce three new operations to the data model in conjunction with segments:

DeclareSegment[segment, file]: opens a segment with the given name, whose data is stored in the given file.

GetSegments[] returns a list of all the segments which have been opened.

SegmentOf[entity or relationship] returns the segment in which a given entity or relationship exists. It may also be applied to relations or domains, since they are entities.

With the addition of segments to the data model, we redefine the semantics of the basic access operations as follows:

1. DeclareDomain and DeclareRelation take an additional argument, namely the segment in which the defined domain or relation will reside. The entity representing a domain or relation now represents data in a particular segment.

2. DeclareEntity and DeclareRelationship are unaffected: they implicitly refer to the segment in which the respective domain or relation was defined. By associating a segment (and therefore a transaction and underlying file) with each relation or domain entity returned to the database client, we conveniently obviate the need for additional arguments to every invocation of the basic operations in the data model.

3. DestroyEntity, DestroyRelationship, GetF, SetF, DomainOf, RelationOf, and Eq are similarly unaffected: they deal with entities and relationships in whatever segment they are defined. Note that by our definition, entities in different segments are never Eq. Also note that nothing in our definition makes a SetF across a segment boundary illegal (i.e. SetF[relationship, attribute, entity] where the relationship and entity are in different segments). Our current implementation requires that special procedures GetFR and SetFR be used on attributes that can cross segment boundaries, see Section 3.

4. DomainSubset and RelationSubset are unchanged when applied to client-defined domains or relations, i.e., they enumerate only in the segment in which the relation or domain was declared. However an optional argument may be used when applied to one of the system domains or relations (e.g. the Domain domain), allowing enumeration over a specific segment or all segments. RelationSubset's attribute-value-list arguments implictly indicate the appropriate segment even for system relations, so a segment is not normally needed unless the entire relation is enumerated.

Note that the data in a segment is stored in an underlying file physically independent from other segments, perhaps on another machine. Introducing a file system into the conceptual data model

may seem like an odd transgression at this point. From a practical point of view, however, we believe it better to view certain problems at the level of file systems. This point of view allows segments to be used for the following purposes:

1. *Physical independence*: Different database applications typically define their data in separate segments. As a result one application can continue to operate although the data for another has been logically or physically damaged. One application can entirely rebuild its database without affecting another, or an application can continue to operate in a degraded mode missing data in an unavailable segment.

2. *Logical independence*: Different database applications may have information which pertains to the same external entity, e.g. a person with a particular social security number. When one application performs a *DestroyEntity* operation, however, we would like the entity to disappear only from that application's point of view. Information maintained by other applications should remain unchanged.

3. *Protection*: Clients can trust the protection provided by a file system more easily than a complex logical protection mechanism provided by the database system. An even higher assurance of protection can be achieved by physical isolation of the segment at a particular computer site. A more complex logical protection mechanism would be desirable for some purposes, but was deemed beyond the scope of Cypress.

4. *Reliability*: Because segment files are physically as well as logically separate, the probability of simultaneous physical failure is lower than for data in the same file. An even higher degree of independence can be achieved using segments at different sites. Replication of data can be provided at the level of segments to provide recovery from failure. Our implementation's file systems do not do this, however.

5. *Performance:* Data may be distributed to sites where they are most frequently used. For example, personal data may reside on a client's machine while publicly accessed data reside on a file server. If the file system provides replication, it can be used to improve performance for commonly accessed data.

Concurrency control may also be handled by the file system, although locks should be at a granularity finer than whole segments (e.g., file pages).

As noted earlier, information about an external entity may be distributed over multiple segments. One or more database applications may cooperate in maintaining the illusion that entities, domains, and relations span segment boundaries. This illusion may be used in at least two ways:

1. Private additions may be added to a public segment by adding entities or relationships in a private segment. The new relationships may reference entities in the public segment by

creating representative entities with the same name in the private segment. An example would be personal phone numbers and addresses added to a public database of phone numbers and addresses: an application program would make the two segments appear to the user as one database.

2. If two applications use separate segments *A* and *B*, they may safely reference each other's data yet remain physically independent. One of the applications may destroy and reconstruct its segment if it uses the same unique names for its entities. If both applications have relationships referencing an entity *e*, and application *A* does a DestroyEntity operation on *e*, the entity and relationships referencing it disappear from application *A*'s point of view, but application *B*'s representative entity and relationships remain.

### 2.9 Augments

In this section, we discuss a more elaborate mechanism for segmenting databases, with more powerful properties. We call these *augments*, because they are "additive" segments as we will see. Entities, domains, and relations are defined independently of augments. An entity with the same name and domain appears as the same entity regardless of particular augments in which data are stored. However, the entity may or may not be defined in a particular augment. Also, relationships continue to be associated with a particular segment.

*NOTE: The ideas in the remainder of Section 2 have not been implemented, they are included for future interest. These features are not discussed again in this report except for Section 5.10, in contrasting data models.*

Augments are intended for two main purposes:

1. *Additive databases*: Information about an entity may be distributed among multiple augments. The database system will make all the augment boundaries invisible, i.e. all the data will appear as if it is one augment from the application program's point of view.

2. *Subtractive databases (versions):* The relations in an augment encode some state of the world the database represents. Updates, representing changes to that state, can be made by adding them in a separate augment. The database may then be viewed with and without the changes (or a number of alternate changes) by adding and removing the augment(s) on top.

The operations on augments themselves appear very similar to those on segments. However, an *ordering* on the open augment list is maintained by the database system. The DeclareAugment call

may specify where in the current list the new augment should be opened: **DeclareAugment[augment, file, previous augment]** opens an augment with the given name, whose data is stored in the given file. It is defined to appear before the given previous augment in the ordering, or at the end if none is given. **GetAugments[]** returns a list of all the augments which have been opened in order. **AugmentOf[relationship]** may be applied to a relationship to determine in which augment it is stored.

With the addition of augments to the data model, we redefine the semantics of the basic access operations as follows:

1. **DeclareRelation** and **DeclareDomain** return a handle representing the relation or domain in all augments. The relation or domain is declared to exist in the augment passed as an argument; the declare procedure may be called additional times to define the same relation or domain to extend into other augments.

2. **DeclareEntity** and **DeclareRelationship** create an entity or relationship, respectively, in the *top-most* augment in the augment list in which the respective domain or relation is defined.

3. **GetF**, **DomainOf**, **RelationOf**, and **Eq** are unchanged, defined in the obvious way. Any entities returned by **GetF**, **DomainOf**, or **RelationOf** are of course augment-independent. **Eq** returns **TRUE** iff the two entities have the same name and their domains have the same name, regardless of whether the entities are in the same augment.

4. **SetF** is defined as before, with two exceptions. Consider the call

   **SetF[r, a, e]**

   where r is a relationship in augment $A$, $a$ is an attribute of **RelationOf[r]** with type $T$, and $e$ is an entity value in domain $D$. This call will cause $e$ to exist in $A$, if it does not already. $D$ must already exist in $A$. Otherwise, the call fails and nothing is created in $A$. (The client may create $D$ and retry the operation).

5. **DestroyEntity** and **DestroyRelationship** destroy the given entity or relationship if there is no namesake of the domain or relation higher in the current augment list. Otherwise, they create an *anti-entity* or *anti-relationship*, respectively, in the top-most client-declared namesake of the domain or relationship. The semantics of **SetF** is, in effect, that of a **DestroyRelationship** followed by a **CreateRelationship** with the attribute changed.

6. **DomainSubset** searches its domain in all open augments, and has the property that an entity in an augment will not appear in the enumeration if an anti-entity with the same name exists in the entity's domain higher in the current augment list. **DomainSubset** never

returns the same entity twice, even though it exists in more than one augment.

7. **RelationSubset** similarly searches its relation in all open augments, and has the property that a relationship in an augment will not appear in the enumeration if an anti-relationship with the same attribute values exists in the relationship's relation higher in the current augment list.

Augments act specially upon relations upon which keys have been defined:

1. If a **CreateRelationship** would create a relationship with the same key value as an existing relationship in the relation in some open augment, then an anti-relationship is automatically created for the existing relationship (in the same augment as the new relationship, the top-most one possible) before creating the new one.

2. If an augment is opened containing relationships whose key values match existing ones in open augments, then anti-relationships for the matching existing relationships are automatically created in the newly opened augment at that time.

The net effect of our definition of augments is that a user or program may make arbitrary modifications to data in an underlying augment, in a fashion which *appears exactly as if a single augment contains all the data.* The underlying augment, however, is completely unchanged. The unmodified data in the underlying augment may concurrently be examined by another user, or appear to be modified through another user's augment. Furthermore, the modifications made by the same user over time are separated and can later be removed.

Note that two augments can be merged in a straightforward way to produce an augment that behaves as the two in the same order in the same place in the current augment list. The augments are merged by combining the elements of the relations and domains in the two augments, discarding entities and anti-entities that match and relationships and anti-relationships that match.

Augments are not related in definition or implementation to the atomic transactions that an implementation of the model also provides. However, the reader might find it informative to think of a transaction as an augment defined on top of the current data, which is automatically merged with the data when the transaction is committed. Transactions thus serve as "short-term" augments. The two mechanisms are useful for entirely different purposes in practice, however.

### 2.10 Views

A *view* is a relation whose relationships are not actually stored. Instead, the primitive retrieval and storage operations on a view invoke procedures defined by a database client who defines the view. Arbitrary procedures may be defined, and views can therefore be used for a variety of purposes:

1. Defining a pseudo-relation in terms of existing relations for convenience: e.g., a **father** view could be implemented in terms of **parent** and **sex** relations that are actually stored.

2. Allowing changes to the logical representation of data (e.g., actually changing the database to store **father** and **mother** relations instead) without changing existing programs written in terms of the older form.

3. Implementing operations on entities in an object-based style by storing tuples in a view. E.g., a **Send** operation on a **Message** entity might be invoked by storing a tuple in a **send** relation with two attributes, the message and the recipient. Any result of such a send operation would be stored in the database (e.g. as a third attribute of the **send** relationship returned). Thus the view mechanism provides a basis for encoding of procedural information in a database.

Views are so-called because they provide a database client a different view of the database than what was originally stored. View definitions and implementations are stored in a database as any other data, being automatically retrieved when required by the database system.

The implementor of a view must define the same operations as needed for any relation. Unlike view mechanisms in some database systems, views may be defined directly in the underlying programming language. In our case, this is Cedar, and the operations the implementation must export are defined as a Cedar interface exported by the view. The view can be dynamically loaded when required at run-time. The operations the view provides are:

1. **RelationSubset** and **CreateRelationship** on the view

2. **DestroyRelationship**, **RelationOf**, **GetF** and **SetF** on its relationships

A view may also be defined at a higher level than the underlying programming language, in a query language; this simplifies the definition of views as well as allowing some kinds of optimizations that need no longer treat a view as a "black box" implementation of an access definition.

### 2.11 Summary

We have introduced the three basic primitives of the Cypress data model: entity values, datum values, and relationships. The basic type checking provided by the model insures that the attributes of relationships are of the proper entity or datum types. Relationship types are called relations.

Entity types are called domains, and a hierarchy of types of domains is permitted.

Entities have names, which uniquely identify them within their domain. Keys may be defined for relations, which consist of one or more attributes whose value must collectively be distinct for every relationship in the relation.

The model provides query operations to enumerate relationships with particular attribute values, or entities with particular names.

The model provides property and translucent attribute operations defined in terms of the basic operations. These operations provide a different perspective on the model, as they allow an entity-centric viewpoint (as in object-based languages) or a relation-centric viewpoint (as in the Relational model), respectively. As a result a client can simultaneously enjoy the advantages of both viewpoints, evaluating relational queries or navigating in a network of objects.

A segment mechanism permits overlapping databases maintained by a variety of applications, or databases distibuted over multiple machines or files by a single application. A more advanced mechanism, augments, duplicates the features of segments and also allows database updates to be encapsulated to maintain different database versions or viewpoints.

The rationale for the Cypress model's features will be discussed further in Section 5; the reader may skip to Section 5 at this point without loss of continuity, or continue with the discussion of our implementation and examples in the next two sections.

# 3. Model level interface

We now describe the Cedar interface to the implementation of the Cypress data model. A knowledge of the Cedar or Mesa programming language (Mitchell et al. [1979]) and the Cedar programming environment is not essential to understanding this section. We will explain Cedar features as they are encountered.

We do assume that the reader is familiar with the basic conceptual data model, i.e., has read the previous section. Our presentation is therefore slightly different in this section: we describe the procedures in the database interface in roughly the order that a client will want to use them in a program. We present types and initialization, schema definition, the basic operations, and then queries.

It should be emphasized that the interface we are about to describe is only one possible implementation of the abstract data model described in Section 2. For example, we have chosen to implement a procedural interface called by Cedar programs, and to do type checking at run-time. We will discuss some of the trade-offs in our choice of interface in Section 6. The introduction of new interfaces, such as a Query level with a compiled access language, will provide a different perspective on the Cypress data model.

## 3.1 Types

In this subsection we describe the most important types in the interface. Less pervasive types are treated at the point where they are first used.

```
Entity:TYPE;
Relship:TYPE;
```

An Entity or Relship is not the actual database entity or relationship; they are *handles* for the actual database objects. All accesses to database objects are performed by calling interface procedures with the handles as parameters. Even comparisons of two entities for equality must be done in this way. The Entity and Relship handles are allocated from storage and automatically freed by the garbage collector when no longer needed.

```
Value:TYPE = REF ANY;
ValueType:TYPE;
Datatype:TYPE;
StringType, IntType, BoolType, AnyDomainType:DataType;
```

Storing Cedar data values in tuples presents several problems. First, since we would like to define a single operation to store a new value into a specified attribute of a Relship (for instance), there must be a single type for all values that pass through this "store-value" procedure. This is the type Value above, represented as untyped REFs in Cedar (a REF is a garbage-collectable Cedar pointer). The DataTypes will be discussed in the next section. Entities, strings, integers, and booleans are the types of values the system currently recognizes and allows as attribute values. More precisely, these four types are Entity, ROPE (the Cedar name for "heavy-duty" strings), REF INT, and REF BOOL. In the case of an entity-valued attribute, an attribute's type may be AnyDomainType or a specific domain may be specified. The latter is highly preferred, as AnyDomainType is a loophole in the type mechanism and limits the kinds of operations that can be performed automatically by the database system or associated tools. We currently provide no mechanism to store compound Cedar data structures such as arrays, lists, or records in a database; the database system's data structuring mechanisms should be used instead. (Cypress query operations such as RelationSubset cannot be composed upon data that appears as uninterpreted bits in the database. We return to this issue in Section 4.)

Note that a Value may be either an Entity or a Datum. Some operations accept any Value, e.g., SetF; others require an Entity, e.g., NameOf. Others may require an Entity from a particular client-defined domain, e.g., a Person. We might think of the hierarchy of built-in and client defined types and instances of values like this:

| Value type hierarchy | Database representative of type |
|---|---|
| Value (REF ANY) | ValueType |
| Datum | DatumType |
| ROPE | StringType |
| INT | IntType |
| BOOL | BoolType |
| Entity | AnyDomainType |
| person Entity | Person domain |
| employee Entity | Employee domain |
| ... other client-defined entities ... | ... other client-defined domains ... |

As Cedar doesn't have a good mechanism for defining type hierarchies or new types for client-defined domains, most Cypress operations simply take a REF ANY or an Entity as argument, performing further type checking at run-time.

### 3.2 Transactions and segments

In this section we describe the basic operations to start up a database application's interaction with Cypress. The client application's data is stored in one or more segments, accessed under transactions. The Cypress system currently runs on the same machine as the client program, however transactions are implemented by the underlying file system which may reside on another machine. Data in remote segments may therefore be concurrently accessed by other instances of Cypress on other client machines.

A transaction is a sequence of read and write commands. The system supports the property that the entire sequence of commands executes *atomically* with respect to all other data retrieval and updates, that is, the transaction executes as if no other transactions were in progress at the same time. Because there may in fact be other transactions accessing the same data at the same time, it is possible that two transactions may deadlock, in which case one of them must be aborted. So the price paid for concurrent access is that programs be prepared to retry aborted transactions.

The database system provides the capability of accessing a database stored on the same machine as the database client, using the Pilot file system (Redell et al. [1979]), or on Alpine file servers (Brown et al. [1983]). We currently permit only one transaction per segment per instance of the database software on a client machine. That is, data in remote segments may concurrently be updated by application programs under separate transactions, but on the same machine transactions are used simply to make application transactions on their respective segments independent. This transaction-per-segment scheme is a major simplification of the Cypress package. In addition, as we shall see presently, nearly all Cypress procedures can automatically infer the appropriate segment and transaction from the procedure arguments, avoiding the need to pass the transaction or segment for every database operation.

Calls to **Initialize**, **DeclareSegment**, and **OpenTransaction** start the database session. A transaction is either passed in by the client, or created by the database package (the latter is just a convenience feature). The operation **MarkTransaction** below forms the end of a database transaction and the start of a new one. The operation **AbortTransaction** may be used to abort a transaction. Data in a database segment may not be read or updated until the segment and transaction have been opened. Clients must decide when to tell the system that a transaction is complete (with **CloseTransaction**), and must be prepared to deal with unsolicited notification that the current transaction has been aborted because of system failure or lock conflict.

The client's interaction with the database system begins with a call to **Initialize**:

```
Initialize:PROC[
 nCachePages: CARDINAL← 256,
 nFreeTuples: CARDINAL← 32,
 cacheFileName: ROPE← NIL ];
```

**Initialize** initializes the database system and sets various system parameters: **nCachePages** tells the system how many pages of database to keep in virtual memory on the client's machine, **nFreeTuples** specifies the size to use for the internal free list of **Entity** and **Relship** handles, and **cacheFileName** is the name of the disk file used for the cache backing store. Any or all of these may be omitted in the call; they will be given default values. **Initialize** should be called before any other operation; the schema declaration operations generate the error **DatabaseNotInitialized** if this is violated.

Before database operations may be invoked, the client must open the segment(s) in which the data is stored. The location of the segment is specified by using the full path name of the file, e.g., "[MachineName]<Directory>SubDirectory>SegmentName.segment". Each segment has a unique name, the name of a Cedar **ATOM** which is used to refer to it in Cypress operation. The name of the Cedar **ATOM** is normally, though not necessarily, the same as that of the file in which it is stored, except the extension ".segment" and the prefix specifying the location of the file is omitted in the **ATOM**. If the file is on the local file system, its name is preceded by "[Local]". For example, "[Local]Foo" refers to a segment file on the local disk named Foo.database; "[Alpine]<CedarDB>Baz" refers to a segment named Baz.segment on the <CedarDB> directory on the Alpine server. It is generally a bad idea to access database segments other than through the database interface. However, because segments are physically independent and contain no references to other files by file identifier or explicit addresses within files, the segment files may be moved from machine to machine or renamed without effect on their contents. If a segment file in a set of segments comprising a client database is deleted, the others may still be opened to produce a database missing only that segment's entities and relationships. A segment is defined by the operation **DeclareSegment**:

```
DeclareSegment:PROC[
  filePath:ROPE, segment:Segment, number:INT← 0,
  readOnly:BOOL← FALSE, version:Version← OldOnly,
  nBytesInitial, nBytesPerExtent:LONG CARDINAL← 32768]
  RETURNS [Segment];

Segment:TYPE = ATOM;
Version:TYPE = {NewOnly, OldOnly, NewOrOld};
```

The **version** parameter to **DeclareSegment** defaults to **OldOnly** to open an existing file. The signal **IllegalFileName** is generated if the directory or machine name is missing from **fileName**, and **FileNotFound** is generated at the time a transaction is opened on the segment if the file does not exist. If version **NewOnly** is passed, a new segment file will be created, erasing any existing one. In this case, a **number** assigned to the segment by the database administrator must also be passed. This number is necessitated by our current implementation of segments (it specifies the section of the database address space in

which to map this segment). Finally, the client program can pass **version = NewOrOld** to open a new or existing segment file; in this case the segment number must also be passed, of course.

The other parameters to **DeclareSegment** specify properties of the segment. If **readOnly = TRUE**, then writes are not permitted on the segment; any attempt to invoke a procedure which modifies data will generate the error **ProtectionViolation**. **nBytesInitial** is the initial size to assign to the segment, and **nBytesPerExtent** is the incremental increase in segment size used when more space is required for data in the file.

For convenience, a call is available to return the list of segments that have been declared in the current Cypress session:

```
GetSegments: PROC RETURNS[LIST OF Segment];
```

A transaction is associated with a segment by using **OpenTransaction**:

```
OpenTransaction:PROC[
 segment:Segment,
 userName, password:ROPE← NIL,
 useTrans:Transaction← NIL ];
```

If **useTrans** is NIL then **OpenTransaction** establishes a new connection and transaction with the corresponding (local or remote) file system. Otherwise it uses the supplied transaction. The same transaction may be associated with more than one segment by calling **OpenTransaction** with the same **useTrans** argument for each. The given user name and password, or by default the logged in user, will be used if a new connection must be established.

Any database operations upon data in a segment before a transaction is opened or after a transaction aborts will invoke the **Aborted** signal. The client should catch this signal on a transaction abort, block any further database operations and wait for completion of any existing ones. Then the client may re-open the aborted transaction by calling **OpenTransaction**. When the remote transaction is successfully re-opened, the client's database operations may resume.

Note that operations on data in segments under different transactions are independent. Normally there will be one transaction (and one or more segments) per database application program. A client may find what transaction has been associated with a particular segment by calling:

```
TransactionOf:PROC [segment:Segment] RETURNS [Transaction];
```

Transactions may be manipulated by the following procedures:

```
MarkTransaction:PROC[trans:Transaction];

AbortTransaction:PROC [trans:Transaction];

CloseTransaction:PROC [trans:Transaction];
```

**MarkTransaction** commits the current database transaction, and immediately starts a new one. User variables which reference database entities or relationships are still valid.

**AbortTransaction** aborts the current database transaction. The effect on the data in segments associated with the segment is as if the transactions had never been started, the state is as it was just after the **OpenTransaction** call or the most recent **MarkTransaction** call. Any attempts to use variables referencing data fetched under the transaction will invoke the **NullifiedArgument** error. A call to **OpenTransaction** is necessary to do more database operations, and all user variables referencing database items created or retrieved under the corresponding transaction must be re-initialized (they may reference entities or relationships that no longer exist, and in any case they are marked invalid by the database system).

A simple client program using the database system might have the form, then:

```
Initialize[];
DeclareSegment["[Local]Test", $Test];
OpenTransaction[$Test];
...
... database operations, including zero or more MarkTransaction calls ...
...
CloseTransaction[TransactionOf[$Test]];
```

### 3.3 Data schema definition

The definition of the client's data schema is done through calls to procedures defined in this section. The data schema is represented in a database as entities and relationships, and although *updates* to the schema must go through these procedures to check for illegal or inconsistent definitions, the schema can be *read* via the normal data operations described in the next section. Each domain, relation, etc., has an entity representative that is used in data operations which refer to that schema item. For example, we pass the domain entity when creating a new entity in the domain. The types of schema items are:

```
Domain, Relation, Attribute, Datatype, Index, IndexFactor:TYPE = Entity;
```

Of course, since the schema items are entities, they must also belong to domains; there are pre-defined domains, which we call *system domains*, in the interface for each type of schema entity:

```
DomainDomain, RelationDomain, AttributeDomain, DatatypeDomain, IndexDomain:Domain;
```

There are also pre-defined system relations, which contain information about sub-domains, attributes, and indices. Since these are not required by the typical (application-specific) database client, we defer the description of the system relations to Section 3.6.

In general, any of the data schema may be extended or changed at any time; i.e., data operations and data schema definition may be intermixed. However, there are a few specific ordering constraints on schema definition we will note shortly. Also, the database system optimizes for better performance if the entire schema is defined before any data are entered. The interactive schema editing tool described in Section 7 allows the schema to be changed regardless of ordering constraints and existing data, by recreating schema items and copying data invisibly to the user when necessary.

All the data schema definition operations take a Version parameter which specifies whether the schema element is a new or existing one. The version defaults to allowing either (NewOrOld): i.e., the existing entity is returned if it exists, otherwise it is created. This feature avoids separate application code for creating the database schema the first time the application program is run.

```
DeclareDomain:PROC [name:ROPE, segment:Segment,
 version:Version← NewOrOld, estRelations:INT← 5] RETURNS [d:Domain];

DeclareSubType:PROC[sub, super:Domain];
```

DeclareDomain defines a domain with the given name in the given segment and returns its representative entity. If the domain already exists and version=NewOnly, the signal AlreadyExists is generated. If the domain does not already exist and version=OldOnly, then NIL is returned. The parameter estRelations is used to estimate the largest number of relations in which entities of this domain are expected to participate.

The client may define one domain to be a subtype of another by calling DeclareSubType. This permits entities of the subdomain to participate in any relations in which entities of the superdomains may participate. All client DeclareSubType calls should be done before declaring relations on the superdomains (to allow some optimizations). The error MismatchedSegment is generated if the sub-domain and super-domain are not in the same segment.

```
DeclareRelation:PROC [
 name:ROPE, segment:Segment, version:Version← NewOrOld] RETURNS [r:Relation];
```

```
DeclareAttribute:PROC [
 r:Relation, name:ROPE, type:ValueType← NIL,
 uniqueness:Uniqueness ← None, length:INT← 0,
 link:{Linked, Unlinked, Colocated, Remote}← yes, version:Version← NewOrOld]
 RETURNS[a:Attribute];

Uniqueness:TYPE = {NonKey, Key, KeyPart, OptionalKey};
```

**DeclareRelation** defines a new or existing relation with the given **name** in the given **segment** and returns its representative entity. If the relation already exists and version=NewOnly, the signal **AlreadyExists** is generated. If the relation does not already exist and **version**=**OldOnly**, then NIL is returned.

**DeclareAttribute** is called once for each attribute of the relation, to define their names, types, and uniqueness. If **version** = **NewOrOld** and the attribute already exists, Cypress checks that the new type, uniqueness, etc. match the existing attribute. The error **MismatchedExistingAttribute** is generated if there is a discrepancy. The attribute name need only be unique in the context of its relation, not over all attributes. Note this is the only exception to the data model's rule that names be unique in a domain. Also note that we could dispense with DeclareAttribute altogether by passing a list into the DeclareRelation operation: we define a separate procedure for programming convenience.

The attribute **type** should be a **ValueType**, i.e. it may be one of the pre-defined types (**IntType**, **StringType**, **BoolType**, **AnyDomainType**) or the entity representative for a domain. For pre-defined types, the actual values assigned to attributes of the relationship instances of the relation must have the corresponding type: `REF INT`, `ROPE`, `REF BOOL`, or `Entity`. If the attribute has a domain as type, the attribute values in relationships must be entities of that domain or some sub-domain thereof. The **type** is permitted to be one of the pre-defined system domains such as the **DomainDomain**, thereby allowing client-defined extensions to the data schema (for example, a comment for each domain describing its purpose).

The attribute **uniqueness** indicates whether the attribute is a key of the relation. If its uniqueness is **NonKey**, then the attribute is not a key of the relation. If its uniqueness is **OptionalKey**, then the system will ensure that no two relationships in **r** have the same value for this attribute (if a value has been assigned). The error **NonUniqueKeyValue** is generated if a non-unique key value results from a call to the **SetP**, **SetF**, **SetFS**, or **CreateRelship** procedures we define later. **Key** acts the same as **OptionalKey**, except that in addition to requiring that no two relationships in **r** have the same value for the attribute, it requires that every entity in the domain referenced by this attribute must be referenced by a relationship in the relation: the relationships in the relation and the entities in the domain are in one-to-one correspondence. Finally, if an attribute's uniqueness is **KeyPart**, then the system will ensure that no two relationships in **r** have the same value for *all* key attributes of **r**,

though two may have the same values for some subset of them.

The **length** and **link** arguments to **DeclareAttribute** have no functional effect on the attribute, but are hints to the database system implementation. For **StringType** fields, **length** characters will be allocated for the string within the space allocated for a relationship in the database. There is no upper limit on the size of a string-valued attribute; if it is longer than **length**, it will be stored separately from the relationship with no visible effect except for the performance of database applications. The **link** field is used only for entity-valued fields; it suggests whether the database system should link together relationships which reference an entity in this attribute. In addition, it can suggest that the relationships referencing an entity in this attribute be physically co-located as well as linked. Again, its logical effect is only upon performance, not upon the legal operations.

```
DestroyRelation:PROC[r:Relation];

DestroyDomain:PROC[d:Domain];

DestroySubType:PROC[sub, super:Domain];
```

Relations, domains, and subdomain relationships may be destroyed by calls to the above procedures. Destroying a relation destroys all of it relationships. Destroying a domain destroys all of its entities and also any relationships which reference those entities. Destroying a sub-domain relationship has no effect on existing domains or their entities; it simply makes entities of domain **sub** no longer eligible to participate in the relations in which entities of domain **super** can participate. Existing relationships violating the new type structure are allowed to remain. Existing relations and domains may only be modified by destroying them with the procedures above, with one exception: the operation **ChangeName** (described in Section 3.4) may be used to change the name of a relation or domain.

```
DeclareIndex:PROC [
  relation:Relation, indexedAttributes:AttributeList, version:Version];
```

**DeclareIndex** has no logical effect on the database; it is a performance hint, telling the database system to create a B-Tree index on the given **relation** for the given **indexedAttributes**. The index will be used to process queries more efficiently. Each index key consists of the concatenated values of the **indexedAttributes** in the relationship the index key references. For entity-valued attributes, the value used in the key is the string name of the entity. The **version** parameter may be used as in other schema definition procedures, to indicate a new or existing index. If any of the attributes are not attributes of the given relation then the signal **IllegalIndex** is generated.

The optimal use of indices, links, and colocation, as defined by **DeclareIndex** and **DeclareAttribute**, is complex. It may be necessary to do some space and time analysis of a database application to choose the best trade-off, and a better trade-off may later be found as a result of unanticipated

access patterns. Note, however, that a database may be rebuilt with different links, colocation, or indices, and thanks to the data independence our interface provides, existing programs will continue to work without change.

If a relation is expected to be very small (less than 100 relationships), then it might reasonably be defined with neither links nor indices on its attributes. In the typical case of a larger relation, one should examine the typical access paths: links are most appropriate if relationships that pertain to particular entities are involved, indices are more useful if sorting or range queries are desired.

B-tree indices are always maintained for domains; that is, an index contains entries for all of the entities in a domain, keyed by their name, so that sorting or lookup by entity name is quick. String comparisons are performed in the usual lexicographic fashion.

```
DeclareProperty:PROC [
 relationName:ROPE, of:Domain, type:ValueType,
 uniqueness:Uniqueness← None, version:Version← NewOrOld]
 RETURNS [property:Attribute];
```

DeclareProperty provides a shorthand for definition of a binary relation between entities of the domain "of" and values of the specified type. The definitions of type and uniqueness are the same as for DeclareAttribute. A new relation relationName is created, and its attributes are given the names "of" and "is." The "is" attribute is returned, so that it can be used to represent the property in GetP and SetP defined in the next section.

### 3.4 Basic operations on entities and relationships

In this section, we describe the basic operations on entities and relationships; we defer the operations on domains and relations to the next section.

A number of error conditions are common to all of the procedures in this section. Since values are represented as REF ANYs, all type checking must currently be done at run-time. The procedures in this section indicate illegal arguments by generating the errors IllegalAttribute, IllegalDomain, IllegalRelation, IllegalValue, IllegalEntity, and IllegalRelship, according to the type of argument expected. The error NILArgument is generated if NIL is passed to any procedure that cannot accept NIL for that argument. The error NullifiedArgument is generated if an entity or relationship is passed in after it has been deleted or rendered invalid by transaction abort or close.

```
DeclareEntity: PROC[
 d: Domain, name: ROPE← NIL, version: Version← NewOrOld]
 RETURNS [e: Entity];
```

DeclareEntity finds or creates an entity in domain d with the given name. The name may be omitted if desired, in which case an entity with a unique name is automatically created. If version is OldOnly and an entity with the given name does not exist, NIL is returned. If version is NewOnly and an entity with the given name already exists, the signal NonUniqueEntityName is generated.

```
DeclareRelship: PROC [
  r: Relation, avl: AttributeValueList← NIL, version: Version← NewOrOld]
  RETURNS [Relship];
```

DeclareRelship finds or creates a relship in r with the given attribute values. If version is NewOnly, a new relship with the given attribute values is generated. If version is OldOnly, the relship in r with the given attribute values is returned if it exists, otherwise NIL is returned. If version is NewOrOld, the relship with the given attribute values is returned if it exists, otherwise one is created. If the creation of a new relship violates the key constraints specified by DeclareAttribute, the signal NonUniqueAttributeValue is generated.

```
DestroyEntity: PROC[e: Entity];
```

DestroyEntity removes e from its domain, destroys all relationships referencing it, and destroys the entity representative itself. Any client variables that reference the entity automatically take on the null value (Null[e] returns TRUE), and cause error NullifiedArgument if passed to database system procedures. After an entity is destroyed, its old name may be re-used in creating a new one.

```
DestroyRelship: PROC[t: Relship];
```

DestroyRelship removes t from its relation, and destroys it. Any client variables that reference the relationship automatically take on the null value, and will cause error NullifiedArgument if subsequently passed to database system procedures.

```
SetF: PROC[t: Relship, a: Attribute, v: Value];
```

SetF assigns the value v to attribute a of relationship t. If the value is not of the same type as the attribute (or a subtype thereof if the attribute is entity-valued), then the error MismatchedAttributeValueType is generated. If a is not an attribute of t's relation, IllegalAttribute is generated.

```
GetF: PROC[t: Relship, a: Attribute] RETURNS [Value];
```

GetF retrieves the value of attribute a of relationship t. If a is not an attribute of t's relation, error IllegalAttribute is generated. The client should use the V2x routines described in the next section to coerce the value into the expected type.

If GetF is performed upon an attribute of a relationship whose value has never been assigned, a special undefined value is returned. The distinguished undefined value depends upon the type of the attribute. For example, for entity-valued attributes, it is NIL. For IntType attributes, it is the largest negative number. Clients of the database system should use the following procedures to deal with undefined values:

```
IsUndefined: PROC [v: Value] RETURNS [BOOL];
MakeUndefined: PROC[d: DataType] RETURNS [Value];
```

IsUndefined takes a value and indicates whether it is undefined, and MakeUndefined produces an undefined value of a particular type, suitable for use with SetF to make a previously defined attribute become undefined.

```
SetFS: PROC [t: Relship, a: Attribute, v: ROPE];
GetFS: PROC[t: Relship, a: Attribute] RETURNS [ROPE];
```

GetFS and SetFS provide a convenient veneer on top of GetF and SetF that provide the illusion that all relation attributes are string-valued. The effect is something like the Relational data model, and is useful for applications such as a relation displayer and editor that deal only with strings. The semantics of GetFS and SetFS depend upon the actual type of the value v of attribute a:

| type | GetFS returns | SetFS assigns attribute to be |
|---|---|---|
| StringType | the string v | the string v |
| IntType | v converted to decimal string | the string converted to decimal integer |
| BoolType | "TRUE" or "FALSE" | true if "TRUE", false if "FALSE" |
| a domain D | the name of the ref'd entity (or null string if v is NIL) | the entity with name v (or NIL if v is null string) |
| AnyDomainType | same, but includes domain: <domain-name>:<entity-name> | the entity with the given domain and name (or NIL if v is null string) |

The same signals generated by GetF and SetF, such as IllegalAttribute, can also be generated by these procedures. The string NIL represents the undefined value. The signal NotFound is generated in the last case above if no entity with the given name is found.

```
NameOf: PROC [e: Entity] RETURNS [s: ROPE];
ChangeName: PROC [e: Entity, s: ROPE];
```

NameOf and ChangeName retrieve or change the name of an entity, respectively. They generate the signal IllegalEntity if e is not an entity.

ChangeName should be used with caution. It is not quite equivalent to destroying and re-creating an entity with the new name but the same existing relationships referencing it. ChangeName is considerably faster than that, and furthermore

entity-valued variables which reference the entity are *not* nullified by ChangeName, though they would be by DestroyEntity. These features should be a help, not a hindrance. However, changing an entity name may invalidate references to the entity from outside the segment, e.g., in another segment or in some application-maintained file such as a log of updates.

```
DomainOf: PROC[e: Entity] RETURNS [Domain];
RelationOf: PROC[t: Relship] RETURNS [Relation];
```

DomainOf and RelationOf can be used to find the entity representative of an entity's domain or a relationship's relation, respectively. The signal IllegalEntity is generated if e is not an entity. The signal IllegalRelship is generated if r is not a relationship.

```
SegmentOf: PROC[e: Entity] RETURNS [Segment];
```

SegmentOf returns the segment in which an entity is stored. It can be applied to domain, relation, or attribute entities.

```
Eq: PROC [e1: Entity, e2: Entity] RETURNS [BOOL];
```

Eq returns TRUE iff the same database entity is referenced by e1 and e2. This is *not* equivalent to the Cedar expression "e1 = e2", which computes Cedar REF equality. If e1 and e2 are in different segments, Eq returns true iff they have the same name and their domains have the same name.

```
Null: PROC [x: EntityOrRelship] RETURNS [BOOL];
```

Null returns TRUE iff its argument has been destroyed, is NIL, or has been invalidated by abortion of the transaction under which it was created.

```
GetP: PROC [e: Entity, als: Attribute, aOf: Attribute← NIL] RETURNS [Value];
SetP: PROC [e: Entity, als: Attribute, v: Value, aOf: Attribute← NIL] RETURNS[Relship];
```

GetP and SetP are convenience routines for a common use of relationships, to represent "properties" of entities. Properties allow the client to think of values stored in relationships referencing an entity as if they are directly accessible fields (or "properties") of the entity itself. See the figure on page 15 illustrating properties. GetP finds a relationship whose from attribute is equal to e, and returns that relationship's to attribute. The from attribute may be defaulted if the relation is binary, it is assumed to be the other attribute of to's relation. If it is not binary, the current implementation will find the "first" other attribute, where "first" is defined by the order of the original calls to DeclareAttribute. SetP defaults the from attribute similarly to GetP, but operates differently depending on whether from is a key of the relation. Whether it is a key or not, any previous relationship that referenced e in the

**from** attribute is automatically deleted. In either case, a new relationship is created whose **from** attribute is **e** and whose **to** attribute is **v**. **SetP** returns the relationship it creates for the convenience of the client. **GetP** and **SetP** can generate the same errors as **SetF** and **GetF**, e.g., if **e** is not an entity or **to** is not an attribute. In addition, **GetP** and **SetP** can generate the error **IllegalProperty** if **to** and **from** are from different relations. **GetP** generates the error **MismatchedPropertyCardinality** if more than one relationship references **e** in the **from** attribute; if *no* such relationships exist, it returns a null value of the type of the **to** attribute. **SetP** allows any number of existing relationships referencing **e**; it simply adds another one (when **from** is a key, of course, there will always be one relationship).

```
GetPList: PROC [e: Entity, to: Attribute, from: Attribute← NIL] RETURNS [LIST OF Value];
SetPList: PROC [e: Entity, to: Attribute, vl: LIST OF Value, from: Attribute← NIL];
```

**GetPList** and **SetPList** are similar to **GetP** and **SetP**, but they assume that any number of relationships may reference the entity **e** with their **from** attribute. They generate the signal **MismatchedPropertyCardinality** if this is not true, i.e. the **from** attribute is a key. **GetPList** returns the list of values of the **to** attributes of the relationships that reference **e** with their **from** attribute. Cedar has LISP-like list manipulation facilities. **SetPList** destroys any existing relationships that reference **e** with their **from** attribute, and creates Length[**vl**] new ones, whose **from** attributes reference **e** and whose **to** attributes are the elements of **vl**. **GetPList** and **SetPList** may generate any of the errors that **GetF** and **SetF** may generate, and the error **IllegalProperty** if **to** and **from** are from different relations.

Note that the semantics of **SetPList** are not quite consistent with the semantics of **SetP**. **SetPList** *replaces* the current values associated with a "property" with the new values (i.e., destroys and re-creates relationships); **SetP** *adds* a new property value, unless the **aOf** attribute is a key, in which case it replaces the current value. The semantics are defined in this way because this has proven the most convenient in our application programs.

Examples of the use of the property procedures for data access can be found in Section 4.3. Properties are also useful for obtaining information about the data schema. For example, **GetP[a, aRelationIs]** will return the attribute **a**'s relation, and **GetPList[d, aTypeOf]** will return all the attributes that can reference domain **d**.

```
E2V: PROC[e: Entity] RETURNS[v: Value];
B2V: PROC[b: BOOLEAN] RETURNS[v: Value];
I2V: PROC[i: LONG INTEGER] RETURNS[v: Value];
S2V: PROC[s: ROPE] RETURNS[v: Value];
```

The x2V routines convert the various Cedar types to Values. The conversion is not normally required for ropes and entities since the compiler will widen these into the REF ANY type Value.

```
V2E: PROC[v: Value] RETURNS[Entity];
V2B: PROC[v: Value] RETURNS[BOOLEAN];
V2I: PROC [v: Value] RETURNS[LONG INTEGER];
V2S: PROC [v: Value] RETURNS[ROPE];
```

The V2x routines convert Values to the various Cedar types. The MismatchedValueType error is raised if the value is of the wrong type. It is recommended that these routines be used rather than user-written NARROWs, as the representation of Values may change. Also, NARROWs of opaque types don't yet work in the Cedar compiler.

### 3.5 Query operations on domains and relations

In this section we describe queries upon domains and relations: operations that enumerate entities or relationships satisfying some constraint.

```
RelationSubset: PROC[
 r: Relation, constraint: AttributeValueList← NIL]
 RETURNS [RelshipSet];

NextRelship: PROC[rs: RelshipSet] RETURNS [Relship];

PrevRelship: PROC[rs: RelshipSet] RETURNS [Relship];

ReleaseRelshipSet: PROC [rs: RelshipSet];

AttributeValueList: TYPE = LIST OF AttributeValue;
AttributeValue: TYPE = RECORD [
  attribute: Attribute,
  low: Value,
  high: Value← NIL -- omitted where same as low or not applicable --];
```

The basic query operation is RelationSubset. It returns a generator of all the relationships in relation r which satisfy a constraint list of attribute values. The relationships are enumerated by calling NextRelship repeatedly; it returns NIL when there are no more relationships. PrevRelship may similarly be called repeatedly to back the enumeration up, returning the previous relationship; it returns NIL if the enumeration is at the beginning. ReleaseRelshipSet should be called when the

client is finished with the query.

The constraint list may be NIL, in which case all of the relationships in r will be enumerated. Otherwise, relationships which satisfy the concatenation of constraints on attributes in the list will be enumerated. If an index exists on some subset of the attributes, the relationships will be enumerated sorted on the concatenated values of those attributes. For a StringType, IntType, or TimeType attribute a of r, the contraint list may contain a record of the form [a, b, c] where the attribute value must be greater or equal to b and less than or equal to c to satisfy the constraint. For any type of attribute, the list may contain a record of the form [a, b] where the value of the attribute must exactly equal b. The Cedar ROPE literals "" and "\377" may be used in queries as an infinitely large and infinitely small string, respectively. The signal MismatchedAttributeValueType is generated by RelationSubset if one of the low or high values in the list is of a different type than its corresponding attribute.

```
DomainSubset: PROC[
 d: Domain,
 lowName, highName: ROPE← NIL,
 searchSubDomains: BOOL← TRUE,
 searchSegment: Segment← NIL]
 RETURNS [EntitySet];

NextEntity: PROC[EntitySet] RETURNS [Entity];

PrevEntity: PROC[EntitySet] RETURNS [Entity];

ReleaseEntitySet: PROC[EntitySet];
```

DomainSubset enumerates all the entities in a domain. If lowName and highName are NIL, the entire domain is enumerated, in no particular order. Otherwise, only those entities whose names are lexicographically greater than lowName and less than highName are enumerated, in lexicographic order. If searchSubDomains is TRUE, subdomains of d are also enumerated. Each subdomain is sorted separately. The searchSegment argument is currently only used if d is one of the system domains, e.g., the Domain domain. It is used to specify which segment to search.

Analogously to relation enumeration, NextEntity and PrevEntity may be used to enumerate the entities returned by DomainSubset, and ReleaseEntitySet should be called upon completion.

```
GetDomainRefAttributes: PROC [d: Domain] RETURNS [AttributeList];
```

This procedure returns a list of all attributes, of any relation defined in d's segment, which reference domain d or one of its superdomains. The list does not include AnyDomainType attributes, which can reference any domain. GetDomainRefAttributes is implemented via queries on the data schema. GetDomainRefAttributes is useful for application-independent tools; most specific applications can code-in the relevant attributes.

```
GetEntityRefAttributes: PROC [e: Entity] RETURNS [AttributeList];
```

This procedure returns a list of all attributes in which some existing relationship actually references e, including AnyDomainType attributes.

### 3.6 System domains and relations

In this section we describe what one might call the *schema schema*, the pre-defined system domains and relations which constitute the data schema for client-defined domains and relations. The typical database application writer may skip this section, since the schema declaration operations defined in Section 3.3 are adequate when the data schema is completely defined and known at the time a program is written. The system domains and relations we describe in this section are most useful for general-purpose tools (e.g., for displaying, querying, or dumping *any* database), where the tools must examine the data schema "on the fly."

As noted earlier, the permanent repository for data describing user-defined data in a database is the database's data schema, represented by schema entities and relationships. Schema entities are members of one of the pre-defined system domains: DomainDomain, RelationDomain, DatatypeDomain, and so on. Every client-defined domain, relation, or attribute contains a representative entity in these domains. Client-defined datatypes are not currently permitted, so the only entities in the DataType domain are the pre-defined IntType, StringType, and BoolType.

The information about the client-defined domains and attributes are encoded by relationships in the database. Domains participate in the system relation dSubType, which encodes a domain type hierarchy:

```
dSubType: Relation;
  dSubTypeOf: Attribute; -- the domain in this attribute is a super-type of
  dSubTypeIs: Attribute; -- the domain in this attribute
```

The dSubType has one element per direct domain-subdomain relationship, it does not contain the transitive closure of that relation. However, it is guaranteed to contain no cycles. That is, the database system checks that there is no set of domains $d_1$, $d_2$, ... $d_N$, N>1, such that $d_1$ is a subtype

of $d_2$, $d_2$ is a subtype of $d_3$, and so on to $d_N$, and $d_1 = d_N$. The **dSubType** may define a lattice as opposed to a tree, i.e. the **sSubType** attribute is not a key of the relation.

The information about attributes is encoded as binary relations, one relation for each argument to the **DeclareAttribute** procedure defining properties of the attribute. The names are easy to remember; for each argument, e.g., **Foo**, we define the **aFoo** relation, with attributes **aFooOf** and **aFooIs**. The **aFooIs** attribute is the value of that argument, and the **aFooOf** attribute is [the entity representative of] the attribute it pertains to. Thus we have the following relations:

```
aRelation: PUBLIC READONLY Relation; -- Specifies attribute - relation correspondence:
 -- [aRelationOf: KEY Attribute, aRelationIs: Relation]
 aRelationOf: PUBLIC READONLY Attribute; -- attribute whose relation we are specifying
 aRelationIs: PUBLIC READONLY Attribute; -- the relation of that attribute

aType: PUBLIC READONLY Relation; -- Specifies types of relation attributes:
 -- [aTypeOf: KEY Attribute, aTypeIs: ValueType]
 aTypeOf: PUBLIC READONLY Attribute; -- the attribute
 aTypeIs: PUBLIC READONLY Attribute; -- domain or datatype of the attribute

aUniqueness: PUBLIC READONLY Relation; -- Specifies attribute value uniqueness:
 -- [aUniquenessOf: KEY Attribute, aUniquenessIs: INT LOOPHOLE[Uniqueness]]
 aUniquenessOf: PUBLIC READONLY Attribute; -- the attribute
 aUniquenessIs: PUBLIC READONLY Attribute; -- INT for Uniqueness: 0 = None, 1 = Key, etc.

aLength: PUBLIC READONLY Relation; -- Specifies length of attributes:
 -- [aLengthOf: KEY Attribute, aLengthIs: INT]
 aLengthOf: PUBLIC READONLY Attribute; -- the attribute
 aLengthIs: PUBLIC READONLY Attribute;  -- INT corresponding to attribute's length

aLink: PUBLIC READONLY Relation; -- Specifies whether attribute is linked:
 -- [aLinkOf: KEY Attribute, aLinkIs: INT]
 aLinkOf: PUBLIC READONLY Attribute; -- the attribute
 aLinkIs: PUBLIC READONLY Attribute;  -- 0 = unlinked, 1 = linked, 2  = colocated
```

The final set of system relations pertain to *index factors.* Each index on a relation is defined to include one or more attributes of a relation. For each attribute in the index, there is an index factor entity. For each index, there is an index entity. Each index factor is associated with exactly one index and exactly one attribute. Indices may have many index factors, however, and an attribute may be associated with more than one index factor, since attributes may participate in multiple indices. The two relations pertaining to indices map indices on to their index factors, and index

factors to the attributes they index:

```
ifIndex: PUBLIC READONLY Relation; -- Specifies the index factors for each index
  -- [ifIndexOf: KEY IndexFactor, ifIndexIs: Index]
  ifIndexOf: PUBLIC READONLY Attribute; -- the index factor
  ifIndexIs: PUBLIC READONLY Attribute; -- index of the factor

ifAttribute: PUBLIC READONLY Relation; -- Specifies attribute index factor corresponds to
  -- [ifAttributeOf: KEY IndexFactor, ifAttributeIs: Attribute]
  ifAttributeOf: PUBLIC READONLY Attribute; -- the index factor
  ifAttributeIs: PUBLIC READONLY Attribute; -- the attribute this factor represents
```

The relations on attributes, index factors, and domains can be queried with the **RelationSubset** or **GetPList** operations. For example, **GetP[a, aRelationIs]** returns the attribute **a**'s relation. **GetPList[r, aRelationOf]** returns the relation **r**'s attributes. **RelationSubset[dSubType, LIST[[dSubTypeIs, d]]]** will enumerate all the **dSubType** relationships in which **d** is the subtype.

As noted earlier, the data schema (attributes, relations, domains, indices, index factors, and relations pertaining to these) may only be read, not written by the database client. In order to ensure the consistency of the schema, it must be written indirectly through the schema definition procedures: **DeclareDomain**, **DeclareRelation**, **DeclareAttribute**, and **DeclareSubType**. Attempts to perform updates through operations such as **SetP** result in the error **ImplicitSchemaUpdate**.

### 3.7 Errors

The Cedar language provides a **SIGNAL** mechanism for returning control to the caller of a procedure when an exceptional condition is identified. When a database system operation invokes an error, the **SIGNAL Error** is generated, with an error code indicating the type of error that occured. The error code is a Cedar "enumerated type:"

```
Error: SIGNAL [code: ErrorCode];
ErrorCode: TYPE = {
        AlreadyExists, -- Entity already exists and client said version = NewOnly
        BadUserPassword, -- On an OpenTransaction
        DatabaseNotInitialized, -- Attempt to do operation without calling Initialize
        FileNotFound, -- No existing segment found with given name
        IllegalAttribute, -- Attribute not of the given relship's Relation or not an attribute
        IllegalValueType, -- Type passed DeclareAttribute is not datatype or domain
        IllegalDomain, -- Argument is not actually a domain
        IllegalFileName, -- No directory or machine given for segment
        IllegalEntity, -- Argument to GetP, or etc., is not an Entity
```

```
    IllegalRelship, -- Argument to GetF, or etc., is not a Relship
    IllegalRelation, -- Argument is not a relation
    IllegalSegment, -- Segment passed to DeclareDomain, or etc., not yet declared
    IllegalString, -- Nulls not allowed in ROPEs passed to the database system
    IllegalSuperType, -- Can't define subtype of domain that already has entities
    IllegalValue, -- Value is not REF INT, ROPE, REF BOOL, or Entity
    IllegalValueType, -- Type passed DeclareAttribute is not datatype or domain
    ImplicitSchemaUpdate, -- Attempt to modify schema with SetP, DeclareEntity, etc.
    InternalError, -- Impossible internal state (possibly bug or bad database)
    MismatchedProperty, -- aOf and als attribute not from the same relation
    MismatchedAttributeValueType, -- Value not same type as required (SetF)
    MismatchedExistingAttribute, -- Existing attribute is different (DeclareAttribute)
    MismatchedExistingSegment, -- Existing segment is different (DeclareSegment)
    MismatchedPropertyCardinality, -- Did GetP with aOf that is not a Key
    MismatchedSegment, -- Attempt to create ref across segment boundary (SetF)
    MismatchedValueType, -- value passed V2E, V2I, etc. not of expected type
    MultipleMatch, -- More than one relationship satisfied avl on DeclareRelship.
    NonUniqueEntityName, -- Entity in domain with that name already exists
    NonUniqueKeyValue, -- Relship already exists with that value
    NotFound, -- Version is OldOnly but no such Entity, Relation, or etc found
    NotImplemented, -- Action requested is not yet implemented
    NILArgument, -- Attempt to perform operation on NIL argument
    NullifiedArgument, -- Entity or relationship has been deleted or invalidated
    ProtectionViolation, -- Read or write to segment not permitted this user.
    SegmentNotDeclared, -- Attempt to open transaction w/o DeclareSegment
    ServerNotFound -- File server does not exist or does not respond
    };
```

In this report, the expression "generates the error X" means that the SIGNAL Error is generated with code=X. Unless otherwise specified, the client may CONTINUE from the signal, aborting the operation in question. Signals should not be RESUMEd except by a wizard who knows the result of proceeding with an illegal operation.

# 4. Application example

This section provides a simple example of the use of Cypress. Section 4.1 introduces the example, a database of documents. Section 4.2 is a discussion of database design: the process of representing abstractions of real-world information structures in a database, somewhat specialized to the data structures available in Cypress. In Section 4.3, a working program is illustrated.

Our example is necessarily short; don't expect any startling revelations on these pages. We will try to consider some of the most common cases, however.

## 4.1 A database application

What are the properties of a well-designed database? To a large extent these properties follow from the general properties of databases. For instance, we would like our databases to extend gracefully as new types of information are added, since the existing data and programs are likely to be quite valuable.

It may be useful to consider the following point. The distinguishing aspect of information stored in a database system is that at least *some* of it is stored in a form that can be interpreted by the system itself, rather than only by some application-specific program. Hence, one important dimension of variation among different database designs is in the amount of the database that is system-interpretable, i.e., the kinds of queries that can be answered by the system.

As an example of variation in this dimension, consider the problem of designing a database for organizing a collection of Mesa modules. In the present Mesa environment, this database would need to include at least the names of all the definitions modules, program modules, configuration descriptions, and current .Bcd files. A database containing only this information is little more than a file directory, and therefore the system's power to answer queries about information in this database is very limited. A somewhat richer database might represent the DIRECTORY and IMPORTS sections of each module as relationships, so that queries such as "which modules import interface Y?" can be answered by the system. This might be elaborated further to deal with the use of individual types and procedures from a definitions module, and so on. There may be a limit beyond which it is useless to represent smaller objects in the database; if we aren't interested in answering queries like "what procedures in this module contain IF statements?," it may be attractive to represent the body of a procedure (or some smaller naming scope) as a text string that is not interpretable by the database system, even though it is stored in a database.

We shall illustrate design ideas with a database of information about documents. Our current facilities, which again are simply file directories, leave much to be desired. The title of a document

on the printed page does not tell the reader where the document is stored or how to print a copy. Relationships between different versions of the same basic document are not explicit. Retrievals by content are impossible. Our goal here is not to solve all of these problems, but to start a design that has the potential of dealing with some of them.

### 4.2 Schema design

Each document in our example database has a title and a set of authors. Hence we might represent a collection of documents with a domain of entities whose name is the title of the document, and an author property specifying the authors:

```
Document: Domain = DeclareDomain["Domain"];
dAuthors: Property = DeclareProperty["author", Document, StringType];
```

Here the authors' names are concatenated into a single string, using some punctuation scheme to allow the string to be decoded into the list of authors. This is a very poor database design because it does not allow the system to respond easily to queries involving authors; the system cannot parse the encoded author list.

Note that in the above definition authors are strings, so anything is acceptable as an author. This weak typing has some flexibility: the database will never complain that it doesn't know the author you just attached to a certain document. However, the system is not helpful in catching errors when a new document is added to the database. If "Mark R. Brown" is mistakenly spelled "Mark R. Browne", then one of Mark's papers will not be properly retrieved by a later search. A step in the direction of stronger type checking is to provide a separate domain for authors.

To represent authors as entities, and to allow a variable number of authors for a document, a better design would be:

```
Document: Domain = DeclareDomain["Domain"];
Person: Domain = DeclareDomain["Person"];
author: Property = DeclareProperty["author", Document, Person];
```

Incidentally, in the last line above we define a property rather than relation for brevity. Instead of the author property declaration we could have written:

```
author: Relation = DeclareRelation["author"];
 authorOf: Attribute = DeclareAttribute[author, "of", Document];
 authorIs: Attribute = DeclareAttribute[author, "is", Person];
```

The property declaration has *exactly* the same effect on the database as the relation declaration, since it automatically declares an author relation with an "of" and "is" attribute. However, the relation is not available in a Cedar Mesa variable in the property case, so operations such as **RelationSubset** cannot be used. Therefore most non-trivial applications will generally not use **DeclareProperty**, but will still use operations such as **SetP** by using one of the relation's attributes. For example, if **joe** is a person entity and **book** is a document entity, one can write:

```
SetP[joe, authorOf, book]
```

or, equivalently,

```
SetP[book, authorIs, joe]
```

We now have one **Document** entity per document, plus, for each document, one **author** relationship per author of that document. Conversely, we have one **Person** entity per person, and one **author** relationship per document the person authored. Of course, these are one and the same author relationships referencing the **Person** and **Document** entities. Figure 4-1 illustrates a few such entities and relationships. Each **author** relationship points to its **Document** entity via the **authorOf** attribute, and to its **Person** entity via the **authorIs** attribute.

Frequently a database application requires some representation of sets or lists, for example to represent the people in an organization or steps in a procedure. Sets and lists are not primitives of the data model per se; sets and lists are normally represented as relations. For example, in our database the set of authors of a particular document is stored via a set of **author** relationships referencing the document. The operation

```
GetPList[book, authorOf]
```

could be used to retrieve this list for some particular book. If we wish to maintain an ordering on this set, e.g., so that the authors of a book are kept in some particular order for each book, we need to use some list representation. In our Cypress implementation **GetPList** (and **RelationSubset**) return relationships in the same order they were created by **SetP**, **SetPList**, or **DeclareRelship**, so that a client may maintain an order by the order of calls. A variant of SetF and SetP is under consideration that allows the client to specify where new relationships should be placed in the ordering of relationships referencing a particular entity. Another alternative, the one conventionally used in the Relational model, is to define another attribute to the relation specify position in the ordering:

```
authorOrder: Attribute = DeclareAttribute[author, "order", IntType];
```

Using an ordering attribute is usually a better solution than depending on the semantics of the Cypress implementation's ordering, as it makes the ordering explicit in the relation. In databases

where a large number of relationships are expected to refer to the same entity, it is also more space efficient in our implementation.

If an **authorOrder** attribute is defined, the client may wish to redefine the **authorOf** attribute so that links (pointers) are not maintained between the **Document** entities and **author** relationships, instead defining a more space-efficient B-tree index on the [authorOf, authorOrder] pair:

```
authorOf: Attribute = DeclareAttribute[
 relation: author, name: "of", type: Document, link: FALSE];

authorIndex: Index = DeclareIndex[author, LIST[[authorOf, authorOrder]]];
```

The Cypress implementation will use this index to process any call of the form

```
RelationSubset[author, LIST[[authorOf, x]]
```

This call to **RelationSubset** will therefore enumerate authors of document **x** sorted by **authorOrder**. Cypress will also use the index in processing for **GetPList[..., authorOf]** as **GetPList** uses **RelationSubset**.

This solution is also somewhat less than perfect, as it depends upon the fact that the Cypress implementation orders relationships when an index exist; but indices are not intended to change the semantics of the operations, only to improve performance. Probably the best solution, if the ordering is important to the semantics of a database application, is to represent a list by a binary "next" relation connecting the entities in an ordering.

Documents have other interesting properties. Some of these, for example the date on which the document was produced, are in one-to-one correspondence with documents. Such properties can be defined by specifying a relation or property as being keyed on the document:

```
publDate: Property = DeclareProperty["publDate", Document, StringType, Key];
```

We are using the convention that domain names are capitalized and relation, attribute, and property names are not capitalized, both for the Cedar Mesa variable names and in the names used in the database system itself. If and when the database system is better integrated with Cedar Mesa, the Cedar and database names will be one and the same.

We might wish to include additional information for particular kinds of documents, for example conference papers. Conference papers may participate in the same relations as other documents. For example, they have authors. In addition, we may want to define relations in which only conference papers may participate, for example a **presentation** relation which defines who presented the paper, and where. We can define a conference paper to be a sub-domain of documents, and

define relations which pertain specifically to conference papers:

```
ConferencePaper: Domain = DeclareDomain["ConferencePaper"];
... DeclareSubType[of: Document, is: ConferencePaper]; ...
Conference: Domain = DeclareDomain["Conference"];
presentation: Relation = DeclareRelation["presentation"];
 presentationOf: Attribute = DeclareAttribute["of", presentation, ConferencePaper];
 presentationAt: Attribute = DeclareAttribute["at", presentation, Conference];
 presentationBy: Attribute = DeclareAttribute["by", presentation, Person];
```

Figure 4-2 illustrates a fragment of a database using this extended design.

The reader will note that we have defined our database schema in the functionally irreducible form described in Section 2.7: i.e., the relations have as few attributes as possible so as to represent atomic facts. This normalization is not necessary in the design of a schema, but often makes the database easier to understand and use, and avoids anomalies in data updates as a result of redundantly storing the same information. Note that the **presentation** relation is an example of a functionally irreducible relation that is not binary. It cannot be decomposed into smaller relations without losing information or introducing artificial entities to represent the presentations themselves.

What other information should be present in our document database? Subject keywords would certainly be useful. Since one document will generally have many associated keywords, we would introduce another relation, say **docKeyword**, to represent the new information. Should keywords be entities? Again there is a tradeoff, but the argument for entities seems persuasive: limiting the range of keywords increases the value of the database for retrieval. The keyword entities could also participate in relationships with a dictionary of synonyms, *Computing Reviews* categories, etc.

This is certainly not a complete design, and the reader is encouraged to fit his or her own ideas into the database framework.

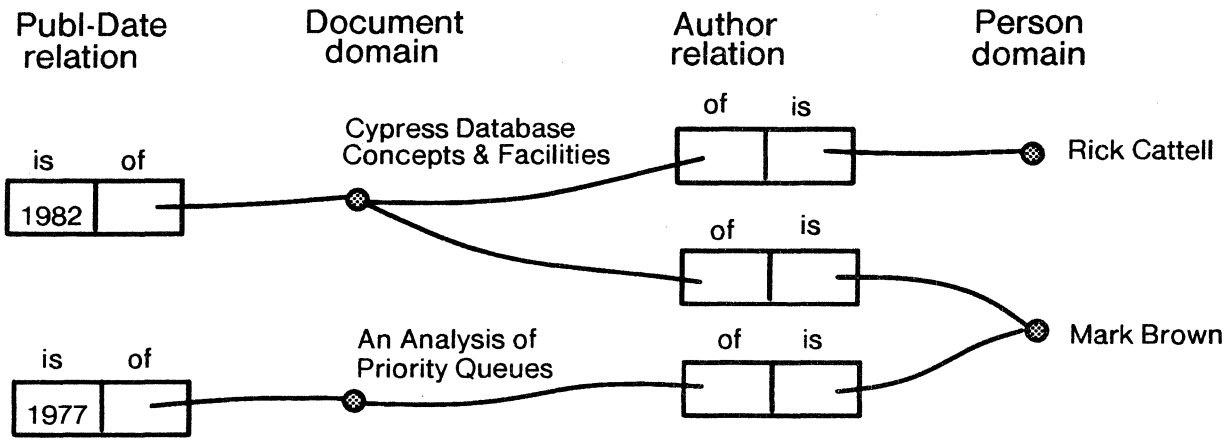



Figure 4-1: A fragment of a simple database of documents

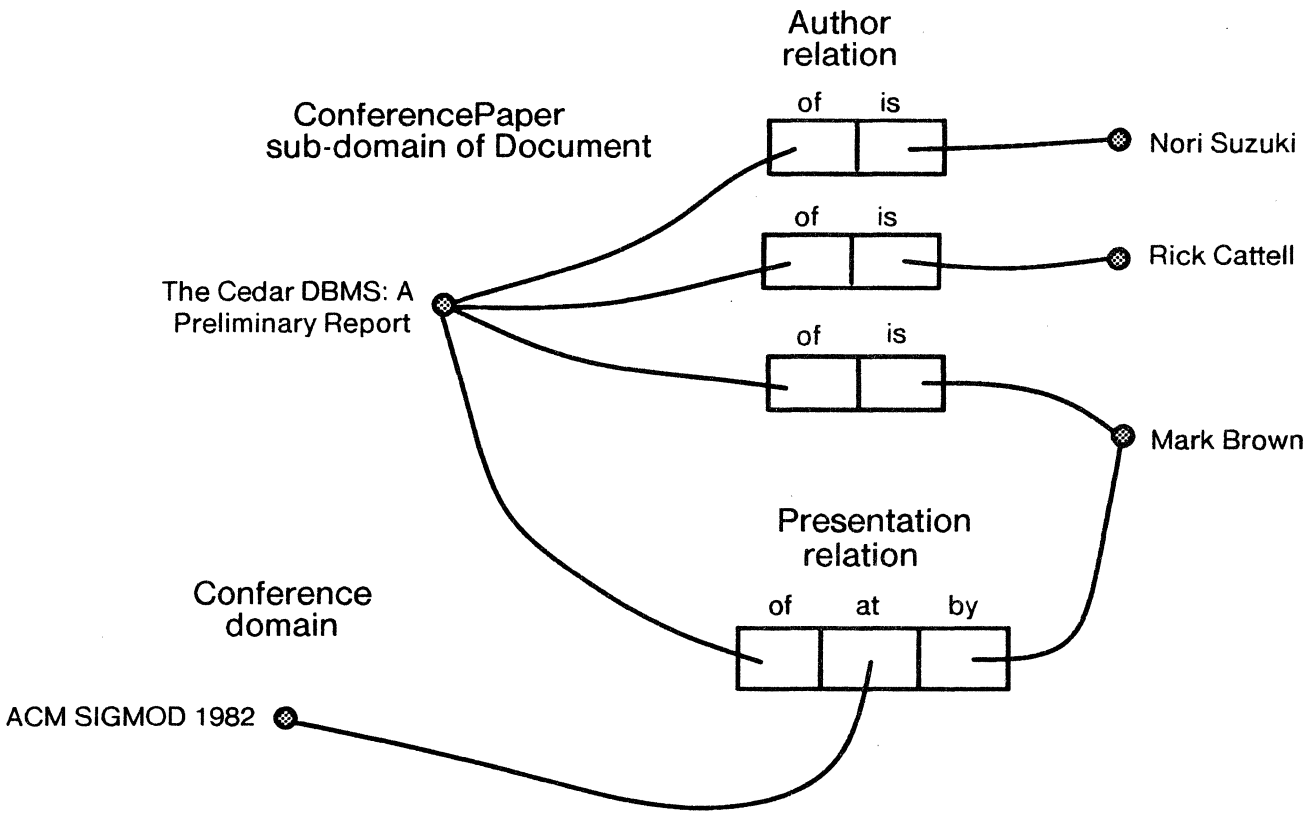



Figure 4-2: A fragment of a document database including conference papers, which can participate in a "presentation" relation.

### 4.3 Example

The following program defines a small schema and database of persons and documents. It illustrates the use of most of the procedures defined in Section 3.

```
DocTestImpl: PROGRAM
 IMPORTS DB, IO, Rope =

BEGIN OPEN IO, DB

tty: IO.Handle← CreateViewerStreams["VLTest1Impl.log"].out;

Person, Conference: Domain;
Thesis, ConferencePaper, Document: Domain;
author: Relation;
 authorOf, authorIs: Attribute;
presentation: Relation;
 presentationOf, presentationBy, presentationDate: Attribute;
publDate: Attribute;
rick, mark, nori: --Person-- Entity;
cedarPaper, cypressDoc, thesis: --Document-- Entity;

Initialize: PROC =
 BEGIN
 tty.PutF["Defining data dictionary...\n"];
 -- Declare domains and make ConferencePapers and Theses be subtypes of Document:
 Person← DeclareDomain["Person"];
 Conference← DeclareDomain["Conference"];
 Document← DeclareDomain["Document"];
 ConferencePaper← DeclareDomain["ConferencePaper"];
 Thesis← DeclareDomain["Thesis"];
 DeclareSubType[of: Document, is: ConferencePaper];
 DeclareSubType[of: Document, is: Thesis];
 -- Declare publDate property of Document
 publDate← DeclareProperty["publDate", Document, IntType];
 -- Declare author relation between Persons and Documents
 author← DeclareRelation["author"];
  authorOf← DeclareAttribute[author, "of", Document];
  authorIs← DeclareAttribute[author, "is", Person];
 -- Declare presentation relation
 presentation← DeclareRelation["presentation"];
  presentationOf← DeclareAttribute[presentation, "of", Document];
  presentationBy← DeclareAttribute[presentation, "by", Person];
  presentationAt← DeclareAttribute[presentation, "at", Conference];
 END;
```

```
InsertData: PROC =
 BEGIN t: Relship;
 tty.PutF["Inserting data...\n"];
 cedarPaper← DeclareEntity[ConferencePaper, "The Cedar DBMS"];
 cypressDoc← DeclareEntity[Document, "Cypress DB Concepts & Facilities"];
 thesis← DeclareEntity[Thesis, "An Analysis of Priority Queues"];
 sigmod← DeclareEntity[Conference, "SIGMOD 81"];
 rick← DeclareEntity[Person, "Rick Cattell"];
 mark← DeclareEntity[Person, "Mark Brown"];
 -- Note we can create entity and then set name...
 nori← DeclareEntity[Person];
 ChangeName[nori, "Nori Suzuki"];
 -- Data can be assigned with SetP, SetF, or DeclareRelship's initialization list:
 t← DeclareRelship[presentation,, NewOnly];
  SetF[t, presentationtOf, cedarPaper];
  SetF[t, presentationBy, mark];
  SetF[t, presentationAt, sigmod];
 []← SetPList[cypressDoc, authorIs, LIST[rick, mark]];
  -- the Cedar notation LIST[ ... ] defines a list
 []← SetPList[cedarPaper, authorIs, LIST[rick, mark, nori]];
 []← DeclareRelship[author, LIST[[authorOf, thesis], [authorIs, mark]]];
 []← SetP[cypressDoc, publDate, I2V[1982]];
 []← SetP[thesis, publDate, I2V[1977]];
  -- the I2V[...] calls needed because Cedar Mesa does not yet coerce INT to REF ANY
 -- Check that thesis can't be presented at conference:
 ok← FALSE;
 t← DeclareRelship[presentation];
 SetF[t, presentationOf, thesis
  ! MismatchedAttributeValueType => {ok ← TRUE; CONTINUE}];
 IF NOT ok THEN ERROR;
 END;

DestroySomeData: PROCEDURE =
 -- Destroy one person entity and all frog entities
 BEGIN flag: BOOL← FALSE;
 tty.Put[char[CR], rope["Deleting Rick from database..."], char[CR]];
 DestroyEntity[DeclareEntity[Person, "Frank Baz", OldOnly]];
 DestroyDomain[Frog];
 END;

PrintDocuments: PROC =
 -- Use DomainSubset with no constraints to enumerate all Documents
 BEGIN
 doc: -- Document -- Entity;
 authors: LIST OF Value;
 es: EntitySet;
 tty.PutF["Documents:\n\n"];
 tty.PutF["Title                    authors\n"];
```

```
  es← DomainSubset[Document];
  WHILE (doc← NextEntity[es]) # NIL DO
   tty.PutF["%g          ", rope[GetName[doc]]];
   authors← GetPList[doc, authorIs];
   FOR al: LIST OF Entity← NARROW[authors], al.rest UNTIL al = NIL DO
    tty.PutF["%g ", rope[GetName[al.first]]]] ENDLOOP;
   ENDLOOP;
  ReleaseEntitySet[es];
  END;

PrintPersonsPublications: PROC [pName: ROPE] =
  -- Use RelationSubset to enumerate publications written by person
  BEGIN
  p: Person← DeclareEntity[Person, pName, OldOnly];
  authorT: --author-- RelShip;
  rs: RelshipSet;
  first: BOOL← TRUE;
  IF p = NIL THEN
   {tty.PutF["%g is not a person!", rope[pName]]; RETURN};
  tty.PutF["Papers written by %g are:\n", rope[pName]];
  rs← RelationSubset[author, LIST[[authorIs, p]]];
  WHILE (authorT← NextRelship[rs]) # NIL DO
   IF first THEN first← FALSE ELSE tty.Put[rope[", "]];
   tty.Put[rope[GetFS[authorRS, authorOf]]];
   ENDLOOP;
  tty.PutF["\n"];
  ReleaseRelshipSet[rs];
  END;


tty.Put[rope["Creating database..."], char[CR]];
Initialize[];
DeclareSegment["[Local]Test", $Test, 1,, NewOnly];
OpenTransaction[$Test];
Initialize[];
InsertData[];
PrintDocuments[];
PrintPersonsPublications["Mark Brown"];
DestroySomeData[];
PrintDocuments[];
CloseTransaction[TransactionOf[$Test]];
tty.Close[];


END.
```

# 5. Data model design issues

A number of decisions were necessary in the design of the Cypress data model described in this paper. A few of these decisions were made arbitrarily because no solution was obviously best. For the most part, however, the criteria of simplicity and utility discussed in Section 1, as we interpret them for the applications we envision, clearly point towards the model we have developed. The decisions that led to the Cypress model chosen are discussed in this section.

## 5.1 Relations and attributes

The Relational model, as described by Codd[1970], was the obvious starting point for the Cypress model. The paper by Codd is easily the most referenced paper in the database literature and the Relational model is widely acknowledged as simple yet reasonably powerful. There have actually been a number of interpretations of "Relational model" in various implementations (such as System R by Astrahan et al. [1976] and INGRES by Stonebraker et al. [1976]). However, all implementations share the most basic idea of a database as a set of relations whose columns are attributes and whose rows are tuples, and most of them share the idea that the tuple attribute values may be strings, numbers, and booleans.

The fundamental type constraint, which states that all the relationships in a relation have the same number and types of attributes, is almost the same in all data models defined since the Relational model. The wording changes somewhat with the introduction of entities or type hierarchies, and some additional constraints are added with the introduction of keys, names, and normalization. We will discuss these minor differences as we contrast the various data model features.

## 5.2 Entities and domains

Codd [1979] introduced the idea of *referential integrity:* that attribute values come from domains such as "People" or "Parts." However, few implementations or theoretical studies of the Relational model carry this idea through to enforcing that persons or parts with unique names actually exist when referenced and that all references to these domains in a database are consistent. Thus, we introduce the concept of an entity to provide the important function of recognizing the objects we are trying to represent in the database. The entity idea frees applications from the integrity checking necessary in the Relational model on strings and numbers that are being used as unique identifiers for objects. We are used to the idea of objects as distinguished from their name, attributes, and relationships to other objects; the introduction of entities thus fulfills the simplicity (understandability) as well as utility criteria.

The introduction of entities has the disadvantage that it tends to "cast in stone" the decision to define a particular real-world object as an entity as opposed to a datum value (or vice versa). For example, if it is decided that the publisher attribute of the author relation is a string datum (the name of the publisher), and later it is changed to be an entity instead, then programs referring to the author relation must be modified. If a client uses the SetFS/GetFS operations, however, it is possible to obtain or even change the publisher without knowing whether it is an entity or datum value. This feature allows an application additional data independence at the expense of ignoring knowledge of entity existence and entity types.

The introduction of entities encodes more knowledge about the structure of the underlying information than the Relational data model. We can use this knowledge to provide better performance, and we do so in the Cypress Database System. For example: (1) entities indicate where physical links are desirable, (2) it is not necessary to store or use artificial numbers to identify objects because unique entity names provide this, (3) entities (atoms) may be stored and manipulated more efficiently than strings or integers (e.g., string matching person-names is more expensive than checking entity equality).

Note that the Cypress data model includes the Relational model as a special case in two senses:

1. If no domains are defined (i.e., all relationship fields are datum values), then we have exactly the Relational model: there is no domain type checking (e.g., that a value is a person's name as opposed to just a string). Any correspondence between strings in one relation and strings in another, and their correspondence to objects, is left to the application programs.

2. The Relational model can be exercised on top of the Cypress data model even in the presence of entities by using only the translucent attribute operations. These operations treat all fields as strings as in the Relational model, but entities and any underlying physical structures required by the implementation are automatically maintained as well. For example, a new entity may automatically be created by using SetFS on a relation tuple. Thus, different users may examine and manipulate the same database as datum values (Relational model), links (graph model), or both (our model).

The introduction of entities to the Relational model is now fairly widely accepted as a good idea, and Chen [1976] is probably most responsible for spreading this notion. Unfortunately, Chen introduces entities in a framework which is more complicated than it needs to be. Entities extend the representational capability of the Relational model in one basic way: they make it possible to distinguish between database items that represent objects and those that represent properties of (or relationships between) objects. Chen uses entities for this basic function, as unique "atoms."

However, Chen and others have combined this function with two other data model features:

1. Treating entities as relationships, by allowing entities to have attributes just as relationships do. Chen does this, and most others have followed his lead.

2. Treating relationships as entities, by allowing attributes of relationships to reference other relationships as well as entities. Chen does not do this, but others (McLeod [1978], Codd [1979], Smith [1977b]) do, as do the older hierarchical and network models.

Combining the basic idea of an entity as an atom and a relationship as a data record can lead to some confusion. In the next two subsections we specifically discuss the merits of (1) and (2), respectively.

### 5.3 Entities as relationships: "attributes" of entities

Chen [1976] and others allow entities to have attributes. That is, they define an entity to have the features of a relationship: entity-valued or datum-valued attributes. This definition divides the information about an entity into two classes: the information that is given by its attributes, and the information that is given by relationships. For example, *age* might be an attribute of a person, while the *projects* he works on might be represented by member relationships between persons and projects. In many cases, the choice is not clear: should *spouse* or *boss* be attributes of persons or relationships between persons? Should *birthday* be a quaternary relation between a person, month, date, and year, or should the birthmonth, birthdate, and birthyear be attributes of a person? These decisions must be made at data definition time and changes later would necessitate changing application programs or providing some kind of translation mechanism.

Applying the simplicity and utility criteria to the entity attribute question suggests that:

1. [Utility] Entity attributes provide a short-hand making application programs and user interfaces simpler: the age of a person can be obtained in one operation instead of two, the extra operation being fetching the relationship. There could also be a performance issue here, but the database system can store relationships physically co-located with the entity about as easily as it can co-locate attributes, so this is not an issue in practice.

2. [Simplicity] Entity attributes introduce additional complexity, both in understandability and the parsimony of the representation. For example, there are now two different mechanisms to represent most facts about entities. There are also indirect, but perhaps more annoying, effects: the syntax and semantics of the query language will be more complicated and/or verbose, if we must handle both cases.

A solution we have not seen in the literature is to allow the best of both worlds, by providing a shorthand to obtain or assign in a single operation the entity properties stored as relationships, but retain relationships as the real underlying logical storage primitive. This line of thinking is the motivation for the introduction of properties in the Cypress model. The result is a model whose utility and simplicity is an improvement on either extreme.

We should note two variations on the idea of allowing entities to have attributes: list-valued attributes and the *characteristic/association* scheme.

A few data models allow list-valued attributes, rather than insisting that only single-valued properties be attributes of entities. For example, the projects a person works on would be an attribute of the person, and the persons working on a project would be an attribute of a project. Note, however, that these attributes of persons and projects are really representing the same fact. By representing the fact as attributes of the entities instead of as relationships the danger of inconsistent data arises (unless yet another feature is added to the model to automatically maintain the interdependencies). Serendipitously, the property mechanism we introduced as a shorthand solves this problem too. The convenient list-valued properties are a veneer on top of the underlying relations, and the consistency of the database is retained at the level of facts stored as atomic relationships.

Another variation on the entity attribute feature is to categorize relations into two types: *associations* and *characteristics* (Pirotte [1977], Codd [1979]). Only *entity values* may be attributes of *associations*. Only *datum values* may be attributes of *characteristics*, except for the first attribute, which is always an entity (characteristic relations relate an entity to datum values that are its "characteristics"). However, because a real-world logical dependency may involve both entities and datum values, it is *also* necessary in these models to allow relationships to be entities, so that the datum values associated with a functional dependency can be expressed as characteristics of the relationship. In some cases, previous authors have further restricted associations and characteristics to be binary relations. There doesn't seem to be any advantage to the association/characteristic scheme, and it was not adopted in the Cypress model.

### 5.4 Relationships as entities: allowing references to relationships

Some data models introduce another feature to the entity/relationship idea: they allow relationships to act as entities in the sense that they can be referenced by other relationships. In the models the author knows of, this is done by eradicating *all* distinctions between entities and relationships. That is, there is only one "object" type of primitive, that can have attributes like a relationship, and can be referenced elsewhere like an entity.

The single object type has the attractiveness of simplicity. One still needs the same operations present before on relationships and on entities, but there is only one type to deal with. Unfortunately, it falls back a bit on the utility criterion, as we lose some features that the entity/relationship distinction was providing us. In addition to the obvious fact that the *user* can no longer tell which data items are thought of as real-world objects, the *system* cannot tell the difference and so cannot:

1. automatically check that objects intended only as relationships are never referenced,

2. perform optimizations that depend on knowing whether a database object can ever be referenced elsewhere, or

3. print the database in a simple linear or human-readable way, as it can when only entities can be referenced and all entities have names (or names can be invented).

Consider the motivation for dropping the entity/relationship distinction. Some relationships, say a *purchase* relationship between a customer, part, and quantity of the part ordered, may act as an entity in and of itself. For example, the purchasing may take part in a *commission* relationship between the purchase, a salesman, and a commission in dollars. If we drop the entity/relationship distinction, this kind of relationship can be added with no change to the existing data schema, as the previously unreferenced purchase relationships can just as easily be treated as entities. Keeping the entity/relationship distinction, such a change would require modifications to database application programs or at least the use of some intermediate interface to shield them from such changes (such as translucent attributes or views).

On the other hand, we can argue that when we choose to treat a relationship as representing an abstract object in and of itself, we should then and only then introduce a domain to represent that kind of event or transaction. Quite simply, that's what an entity is for: it says that we are thinking of this database item as an object, that can participate in relationships.

In a more practical and perhaps more convincing vein, not allowing explicit references to the relationships in a database permits a much simpler query language, since it need not deal with nesting of relations. Relationships referencing relationships provide a potential spaghetti of interconnections. In particular, as noted in earlier sections, we may still use a relational query language in our data model, despite the introduction of the entities, names, domains, subtypes, and so on. This compatibility is possible exactly because of the dichotomy we introduced between relationships (objects that may reference other objects but may not themselves be referenced), and entities (objects that may be referenced but may not reference others). We have a simple, two-level structure.

In applying the simplicity and utility criteria here, the choice is difficult. Our choice, to keep the entity/relationship distinction, was motivated by the observations that (1) dropping the distinction results in no fewer operations either in the database interface or the application program, it merely condenses two types into one; (2) there is a loss of some integrity checking and data semantics by dropping the distinction; (3) a linear and human-readable notation of entities and relationships, using entity names, is possible; and (4) the query language is simplified by permitting only one information-carrying element, relations.

There is a variation on dropping the entity/relationship distinction that allows references to relationships only in a non-cyclic hierarchical fashion; this middle ground seems to be more complex and/or less powerful than either of the extreme positions, however.

### 5.5 Lists and sets

A few data models permit set values for relationship attributes. We discussed this in the previous subsection in the dual form of multi-valued attributes of entities. Both have the same drawback in maintaining consistency of the database. Both violate Codd's [1970] first normal form designed to avoid such inconsistencies.

On the other hand, we have found it useful in database applications to have some mechanism to represent the concept of a *list*, an ordering upon database entities or relationships. Almost none of the data models of which the author is aware provide any help in representing the concept of a list. The standard way to represent ordering in the Relational model is by adding another attribute to the relation, specifying the ordinal position of the tuple. A list thus constructed can only be enumerated efficiently if an index is built upon the attribute, and it is quite awkward to insert new tuples in the middle of the list.

We do not claim to have solved the list representation problem in the Cypress data model, either. However, in practice we have used the list representation our *implementation* provides: the order maintained on references to an entity. When a **RelationSubset** is executed with an entity-valued attribute-value pair **[a, e]** as the constraint, the tuples that reference the entity are returned in a particular order. This ordering is guaranteed by the group lists we will describe in Section 6. It is the same order as the entities were specified in the call to **SetPList**, or in the order they were created if **SetP** or **DeclareRelship** calls were used. We are experimenting with an optional extra argument to **SetF** which specifies where in the list the new tuple should be inserted when a new reference to an entity is established:

```
SetF: PROC[t: Relship, a: Attribute, v: Value, after: Relship← NIL];
```

Using this mechanism, our applications can define binary *member* relations as connectors to entities that represent ordered (or unordered) sets. Elements can easily be added and removed from a set with SetF or SetP, and the sets can easily be enumerated with RelationSubset.

However, we have not advertised this feature in the description of the data model or the DBModel interface because it is only a tentative solution. Not only does this solution depend upon the implementation of the entity references via linked lists, but the feature is not accessible at all from a higher-level Relational query language. More work is badly needed, to extend data models and query languages to cover the concept of ordering in a clean and integrated way.

### 5.6 Entity names

The introduction of entity names to the data model is a comparatively easy choice. They easily satisfy the criteria of simplicity and utility. What is less clear is the form in which they should be introduced.

For the applications we have encountered, nearly all real-world objects have human-meaningful names that are unique within their domain or can easily be made to be so: people, organizations, articles, programs, calendar years, or electronic messages. The natural real-world name is preferred to an internally generated identifier for the object, if the real-world name is guaranteed to be unique, because the name is the way in which a user will typically identify an object to the computer and represent the object outside the database.

A number of applications use names that logically have multiple parts. The name of a person, for example, might consist of the concatenation of a first, middle, and last name. Some applications have *dependent* names. A dependent name is a multiple-part name one of whose parts is another entity. The name for a file system subdirectory, for example, might be a two-part name consisting of the entity for its super-directory plus the string name of the subdirectory. The name of a wine, e.g., a 1979 Sebastiani Zinfandel, has three parts: the vintage year, the winery entity, and the variety.

The ubiquity of multi-part and dependent names easily satisfies our utility criteria. They probably also satisfy our simplicity criteria, but this is less obvious. Closer examination reveals that the implementation of names must be reasonably complex in order to satisfy typical applications' needs. Different separators for the name parts and different conventions for their combination and sort order may be required: e.g., "Lastname, FirstName Initial" for persons, or "<SuperDirectory>SubDirectory>" for file system directories. Also, the data schema mechanism

must define the name in terms of independent existing relations. That is, one probably does not want to think of the name components as "attributes" of an entity in the sense that relationships have attributes. The relation between wines and wineries or the relation between organizations and their parent organizations are queried independently but are also used in derivation of wine and organization names.

Our conclusion is that the data model should allow for a procedural specification of the name derivation so that names can automatically be updated when data change, rather than allowing some fixed name derivation primitives. This allows an arbitrary definition of names, just as views may allow arbitrary relation definitions. It also removes the need for the database system to understand names at all, since it treats the name function as a "black box." Until storage of procedural information is easier in the Cedar programming environment, however, the name function has been omitted from the implementation.

One could imagine applications that use integers or dates as entity names rather than strings. Again, this variation could be handled by a procedural specification of the name derivation, and we found insufficient demand to explicitly incorporate this into Cypress.

Real-world objects can have more than one name, so it might be reasonable to allow this in the data model as well. Multiple names were omitted from the Cypress model since the user can build an *alias* relation on top of the system that is checked whenever names are looked up. The only common example of aliases we have encountered are for person entities, so name aliasing does not satisfy the utility criteria. However if multiple names are used widely it does make more sense to augment the model. In that case it is still desirable for every entity to have a *primary* name to allow for correct operation of GetFS and SetFS.

Another more difficult design decision for names in the data model is whether to allow null names for entities (Codd [1979], Kent [1978], Date [1981]). Without the guarantee that every entity has a unique identifier, a number of database operations become more complex in both semantics and implementation: expression of queries, printing of entities, dumping of databases, cross-database references (augments), data entry, and maintaining the correspondence between relational and entity-based database access (GetFS, SetFS). Unfortunately some applications do not require names. For example the transistors in an LSI layout may be represented as entities accessible only through their relationships to connected entities in the layout. The best solution seems to be to require that entities have names, but provide a default system facility for generating unique names. The GENSYM facility for generating unique atoms in LISP does this. The DeclareEntity procedure in our implementation acts similarly. Thus the data model's invariants remain simple but add no additional complexity to application programs not requiring names.

### 5.7 Keys, normalization, and dependencies

Any kind of key mechanism is only an approximation to a description of arbitrarily complex dependency relationships between entities and datum values involved in relationships. Functional and multi-valued dependencies and their implications for relational normalization have been extensively studied in the literature (Codd [1970], Fagin [1977, 1981], Sadri & Ullman [1980]). Keys have the advantage of simplicity over a more general, open-ended scheme requiring constraint checking on a list of axioms on each database update: 80% of the checking can be handled with 20% of the complexity. We are forced once more to apply our utility constraint: how complex a key mechanism would most database applications use? Uniqueness of entity names in domains? Single-attribute primary keys for relations? Multiple-attribute primary keys? Optional keys (unique if present)? We include all of these mechanisms in the Cypress data model, but not arbitrary assertions.

Normalization is typically not enforced by a database system. The database administrator attempts to design the data schema in a manner to minimize various anomalies, through normalization. The subject of database normalization, as noted in the previous paragraph, has been extensively studied in the literature. It is my opinion that arbitrarily complex kinds of normalization and dependencies will continue to be discovered. The easiest solution, therefore, is to take the extreme position: adopt the convention of normalizing databases as much as possible at the conceptual level. We define functionally normalized to mean that relationships in the model are single irreducible facts. The result in practice is that most relations are binary; the physical database representation may in fact join relations in storage for efficiency, but this optimization is not visible at the client's conceptual level.

Hall et al. [1976], Biller [1979], and Schmidt & Swenson [1975] argue that the irreducible relational form we are advocating is preferable to joining unrelated functional dependencies into a single relation. There are several arguments: (1) this is easier to understand because the relational representation matches the real-world dependencies, (2) if there is efficiency in actually joining them, this can be achieved by storing the join in the physical representation and defining the real relations by views that are projections of this underlying relation, (3) this representation is more "restrictive" than any normal form that could be derived. The irreducible form has therefore been used by convention, although not enforced, in the Cypress system applications. Features of the database system and associated tools make this form easy to use.

Our definition of irreducibilty is *not* the same as irreducibility to relations whose join can reproduce the original relation, as in Hall et al. [1976] but rather what Biller [1979] calls s-irreducibility. Biller proves that our definition is more restrictive.

We backed off one step from irreducibility to what we called "functional irreducibility:" in this

form we allow two or more relationships to be combined into a single relationhip when they must logically all be present or all be absent for each external (real world) entity. In practice we find that functional irreducibility differs from irreducibility only when we separate components of a single real world value into its constituent parts. For example, the *birthday* relation given as an example of a functionally irreducible 4th-order relation in Section 2.7 would have been a binary relation had the month, day, and year been combined into a single "date" value. Kent [1982] discusses the issue of combining or separating components of values or entity names.

The last topic we cover in this section is "existence dependencies" between entities and relationships, in particular the fact that DestroyEntity is defined to include the destruction of relationships that reference the entity. One could imagine other semantics:

1. When an entity is destroyed, the attributes of relationships that reference it could become NIL.

2. We could disallow the destruction of entities altogether, destroying only relationships.

3. When two database applications share entities of a particular domain, e.g., both a message system and a phone directory share information about people, deletion of an entity by one application should only delete relationships pertaining to the entity for that application, not the entity itself.

Alternative (1) was ruled out on the basis of our utility criterion. For the database applications we have and envision, the desired semantics is to destroy all information about an entity when the entity is destroyed. Alternative (2) is attractive, as it simplifies the model by reducing the number of operations we must understand and implement. However, it was also ruled out on the basis of our utility criterion; there are cases where an application program really needs to remove an entity completely from the database, e.g., when old data is purged. Furthermore, the author knows no simple implementation for this alternative not requiring "garbage collection" of entities created but no longer used.

Alternative (3) is an important issue, as the support of applications with overlapping database schemas is crucial. Of course, applications that share domains will have to be in agreement as to which relations belong to whom anyway, and could delete only their own data. But a more powerful mechanism that could delete the relevant data in one operation is desirable. The solution to that problem, in the opinion of the author, is to use the data segmentation mechanisms we consider further in Section 5.9. These mechanisms not only allow physical independence of the applications, but deletion of the entity representative of a real-world object in one segment has no effect on the entity or its relationships in another segment.

### 5.8 Generalization and type hierarchies

A wide variety of mechanisms have been forwarded in the literature for specifying types and subtypes of data objects, or, equivalently, allowing the same object to participate in more than one class of relationship. These mechanisms have variously been called type hierarchies, type hierarchies with multiple inheritance, type lattices, roles, and flavors.

There are two major data modelling questions to address: how powerful a type-subtype mechanism is desirable, and whether it should apply to domains, relations, and/or datatypes. Both of these choices are difficult, and were made on the basis of the applications and schemas of the applications we envision.

At least some sort of type lattice ranks high on our utility criterion, as many applications define relations that are constrained to a subset of the entities of a type to which more general relations also apply. A database of documents for example, would separate those documents that are individual works (books, articles, technical reports) from those that are collections (journals, conference proceedings, or collected books).

Although we can construct examples where more complex type mechanisms such as roles or flavors are desirable, for example treating a person entity as either a homeowner or employee or both, none of the applications we envision would use this generality and the more complex mechanisms are harder to understand as well as more complex in their implementation. The addition of multiple supertypes to the basic type hierachy mechanism, on the other hand, is a relatively simple concept to understand and implement, and handles most data schemas the more complex mechanisms handle. Thus we arrive at the type hierarchy with multiple supertypes.

Yet another variation on the type-subtype mechanism is to define subtypes by a *predicate* computed at run-time from data in the database (McLeod [1978]). For example, an entity would be defined to have type mother if it has type person and also has a parent relation to one or more children. Computing types at run-time unfortunately introduces much of the complexity of arbitrary constraint satisfaction tests earlier rejected in our database normalization disucssion. It also provides more power than called for by our applications and the utility criterion. Subtypes computed by predicate were therefore omitted from the Cypress data model.

In their well-known paper on generalization (type hierarchies), Smith and Smith [1977] apply the concept to all kinds of database objects, and we would naturally apply it to relations, domains, and datatypes. This idea certainly satisfies our simplicity criterion, as it makes sense to apply generalization universally in the data model if at all. In our implementation, however, only type hierarchies of relations are not allowed: these do not seem particularly useful to applications. Since user-defined datatypes are not currently permitted, there is no need for a subtype mechanism for them, either. However, a subtype mechanism for datatypes *does* satisfy our simplicity and utility

criteria when user-defined datatypes are defined. The hierarchy of relation types is difficult to implement when multiple supertypes are allowed and attributes of relationships are stored in a physically contiguous memory, because the multiple supertypes may define overlapping incompatible fields. Furthermore, the hierarchy of relation types seems to be of less utility.

The effect of subdomains could be achieved by allowing relation attributes to accept one of a list of types instead of by using the predefined SubType relation to define a lattice of types, specifying the [one] least common domain for each attribute. These two alternative representations do not differ much in flexibility of the type mechanism. The former might be better if we found it necessary in the latter scheme to define a large number of "artificial" intermediate types in order to achieve the type constraints we desire. However this has not been the case in our experience. The single type for attributes might be simpler to understand, but the argument is not a convincing one. Therefore the choice we made, to use the Subtype relation, was arbitrary.

### 5.9 Access primitives

Most database management systems (INGRES and System R, for example) provide an interface to tuples (relationships or entities) at a coarser grain than ours. They do not allow tuple-valued variables in a client program, on which operations such as GetF and SetF can be invoked to extract or assign particular values. Instead tuples are explicitly *copied* from a database into buffers allocated by the client, and explicitly copied back on an update. The difference between our approach and theirs is subtle, but it may have a major impact on the design of an application. Because the "handles" for the tuples in our scheme are garbage-collected, the client need not keep track of the number, logical source, allocation, or size of variables referencing database objects.

One additional complexity *is* introduced by the granularity of database access in Cypress. Specifically, the problem is not a result of our choice of "call by reference" over the "call by value" in other systems, but that all of the updates to a particular tuple are not necessarily encapsulated in a single database call. In examining the interface to the implementation described in Section 3.4, the reader will note that relationship attributes can either be assigned by calling DeclareRelship with a list of attribute values, or by successive calls to SetF. The former is preferable, as it sometimes allows the database system to more efficiently update the data. In addition, the latter complicates the implementation and/or the rules for database system use, for example in the use of the *surrogate relation* optimization we discuss in Section 6. For now, however, we have retained SetF in the implementation as a convenience to the client.

In addition to the call by reference/value distinction, INGRES and System R differ from our design in providing a higher level of access to database clients: a query language. Of course, we plan the addition of a Query level to Cypress (or multiple different query levels). The difference remains,

however, that we are willing to have clients access the database at the lower "navigational" level as well. Our introduction of entities into the Relational model makes a navigational interface compatible with concurrent use at a higher level. The introduction of entities also makes possible query languages of a different style, e.g., more like predicate calculus, or using a "chain" of domains with relations as connectors.

### 5.10 Views, segments, and augments

There is little question that views are quite useful and also relatively easy to understand, and thus satisfy our utility and simplicity criterion. A more difficult question in the design of views is the power of the view definition language: should an arbitrary programming language be allowed, or a more restricted specialized query language? We have essentially avoided this issue, by separating the definition of the query language from the definition of the data model itself. A particular choice will have to be made in the second phase of development of the Model Level of Cypress.

In addition to the utility of relational views as discussed in the literature, e.g., for data independence, they can be used to encode operations on entities in an object-oriented form. The operations of hiring or firing an employee or sending a message can be invoked by storing a tuple into a relation whose attributes are the arguments of this function. For example, if the operation of sending a message takes two arguments, the message and the recipient, the actual operation would be invoked by storing a tuple in the Send view (relation) with the appropriate message entity and recipient entity as attributes.

In general, we have taken a reasonably conservative philosophy in the design of the Cypress data model, integrating ideas that have separately been studied theoretically if not implemented in studies in the literature. This philosophy increases the probability of a practical implementation of the model. The introduction of augments is an area where we have violated the conservative philosophy, however, because the functionality of augments seems so important to applications.

Augments and segments have roots in the segments of System R (Astrahan et al. [1976]). Our segments are slightly more powerful than theirs, as we allow references across segment boundaries; augments are of course more powerful still. System R segments cannot be used for encapsulating updates to data or to combine private and public data. Relations cannot cross System R segment boundaries.

The use of augments to represent versions of data makes them similar to that of the "layers" of Goldstein and Bobrow [1980] and the "hypothetical databases" of Stonebraker [1980]. None of these mechanisms have been implemented on a database system scale at the time of this writing. The semantics of layers, hypothetical databases, and augments differ slightly at the conceptual level. Augments have the additional feature that they have semantics at a *physical* level: they can be used

for separating data that is physically independent in the face of software and hardware failures, and the data may be distributed over machines.

Transactions provide a mechanism to encapsulate updates to a database. A natural approach to providing the capabilities of augments in a database system might be to transform transactions into permanent, full-fledged objects. Perhaps in terms of the simplicity criterion, the elaboration of transactions has some merit. However, the implementation of long-term and short-term encapsulation of data updates must be quite different and the system would at least require some hint as to which physical model to use for a particular update. We chose in the Cypress data model to make the permanent versus temporary distinction show through to the client, because there seems to be little or no utility in avoiding the distinction. The applications we envision require transactions and/or augments, but not interchangeably.

We currently chose to implement segments rather than augments. A segment allows new data (entities or relationships) to be inserted, possibly referencing entities in other segments. However, data may not be deleted or modified in a different segment than it resides. Furthermore, applications must simulate the effect of relations crossing segment boundaries where desired (using the "namesakes" correspondence). Segments handle many of the applications we envision, combining orthogonal databases from different applications. A few applications require augments, however (e.g., a database of different versions of systems). Others are simplified by augments (e.g., several databases of people with overlapping information).

Augments add little or no conceptual complexity: they simply make an entity with a given name unique throughout instead of existing on a segment basis. Subtractive augments also have some utility even where they currently seem unnecessary: e.g., we might want to make personal modifications rather than simply additions to a public database. We conclude that both the utility and simplicity criteria are satisfied by augments. The *implementation* of augments is considerably more complex than segments, and so was deferred.

### 5.11 Summary

In this section we have explained the rationale for our choice of data model. The criteria for our selection are simplicity and utility, viewed from the perspective of existing or envisioned database applications. Surprisingly, the choice of data model features in Cypress has been reasonably clear cut. Examining the needs of actual applications rather than postulating *potential* uses has greatly simplified the choice of model.

We started with the relational data model, adding the concept of entities. Entities insure referential integrity: the constraint that relations be between objects of specified types. We introduce entities

in a way different from most other data models. Some models treat entities as relationships: that is, they allow entities to have fields just like relationships. We rejected this approach as it provides two redundant representations of information and thereby violates the simplicity criterion as we define it. Some models treat relationships as entities: that is, they allow references to relationships from other relationships. We rejected this approach because it results in a network "spaghetti" to which a high-level query language cannot be applied.

We augmented the basic entity-relationship model with features of high utility and little additional complexity: relational keys, unique entity names, segments, and a hierarchy of types. We also provide a limited mechanism to represent existence dependencies between data: relationships are deleted when an entity they reference is deleted, but relationships and entities in separate segments are independent. The basic features of Cypress are present in one or more other data models, although this is probably the first model combining all of these ideas.

Other data model features appear to have high utility, but more work is needed to provide a useful implementation of such features. Augments fall in this category. More complex (multi-part, multi-type) entity names would also be useful, but we know no sufficiently general and efficient implementation. Cypress has a limited mechanism for representing lists and sets using relationships to represent membership, but a better mechanism would be desirable.

The Query level of Cypress will provide relational views and a higher level access language. A fair amount of research work already exists in these areas, and Cypress provides the same basis as other database systems for query and view construction. The Query level has been deferred for now, however.

## 6. Data model implementation issues

In this section we describe important aspects of our implementation of the Cypress data model defined in the previous sections. In the first subsection, we give an overview of the Cypress design and the storage primitives with which we are dealing. Then we discuss a number of specific optimizations we have found useful in the implementation. In the last subsections, we describe the implementation of each of the major operations in more detail and summarize our results.

The Cypress DBMS runs on personal computers, in the Cedar programming environment developed in the Computer Science Laboratory. It is written in the Cedar language, a descendent of Mesa (Mitchell et al [1979]) with changes aimed at making Cedar particularly convenient for developing prototype systems.

Cypress is built in four levels: the Model level, which implements the data model; the Storage level, which provides basic storage allocation and access structures; the Segment level, which manages the database system's virtual memory; and the File level, which provides transaction-based file access.

The File level can be one of a variety of file systems that provide page-level transaction-based access to disk files. We currently use the Pilot system for files on the database client's machine, and the Alpine file server for remote files. In the latter case, the Model, Storage, and Segment levels run on the user's personal machine, and the File level on an Alpine file server.

The Segment level may communicate with the File level by local or remote procedure call. The File level provides procedures to:

1. open, close, and abort transactions,

2. open disk files under transactions, and

3. read and write file pages.

The Segment level provides an LRU cache of database pages from the various segments, reducing network traffic when the File level resides on a different machine. The Segment level implements the segment abstraction, including a mapping from 32-bit database addresses to underlying file pages.

The lower levels of Cypress (the Segment and File levels) will not be of central concern to us for the purposes of this report. In the next subsection, we describe the Storage level interface, and in the remainder of Section 6 we discuss the Model level implementation.

### 6.1 Storage level structures

The underlying Storage level provides the physical data structures used by the Model level: *recordtypes*, *group lists*, and *indices*. There is little that is particularly new in the Storage level, it uses reasonably well-understood mechanisms (the design is much like the RSS level of System R, Astrahan et al [1976]).

*Recordtypes* are the basic storage mechanism for data. A recordtype is a set of records with fields that can be (1) fixed length blocks of words, (2) variable-length byte strings, or (3) references to other records. Recordtypes are manipulated using *recordtype* and *field* handles that describe what the Storage level needs to know to perform an operation on a recordtype or one of its fields, respectively. (Specifically, we need to know the byte length of records in a recordtype, and the byte position and size for its fields.)

Both entities and relationships are represented as records. Entities are represented as records whose first field is the name. Relationships are normally represented as records whose fields are the attributes of the relation. Each domain or relation maps to a particular recordtype.

Every record has a unique 32-bit tuple identifier or *TID*, which uniquely defines it. The Cypress TID for a record contains its segment number, the page number where it is stored in that segment, and its record number on that page. TIDs can be used to find a record relatively efficiently, normally in one page access. The record number on a page is mapped to a record through an array at the beginning of the page. A record could be moved to another page without changing all references to its TID, simply by entering a forward reference in this array, to its new location on an overflow page. We do not currently use this feature, however.

*Groups* are doubly-linked lists maintained through records, also known as *parent-child sets* (System R). A "parent" record resides at the head of the group. The other records in the list contain three pointers:

next: the TID of the next record in the group
prev: the TID of the previous record in the group
parent: the TID of the parent record

Groups are declared by defining a *group field* of a recordtype. Every record referenced by one of these group fields is the parent of a group list containing all records which reference that record in that group field.

The Model level currently uses groups as follows. If an attribute **a** is an entity-valued attribute declared with **link = Linked** (see **DeclareAttribute**), it is defined as a group field at the Storage level. Thus, a group list is maintained for each entity **e** referenced by attribute **a** of one or more records in

a's relation. Those records (the relationships which reference e) are linked together, with the entity e at the head of the list. For example, an entity of type "person" might be at the head of a number of group lists, one for the first attribute of the "author" relation, two for the first and second attributes of the "friend" relation, one for the second attribute of the organization-member relation, and so on. We can quickly find all the author relationships referencing a particular person by following the group list associated with the first author attribute. Group lists are updated by the SetF operation on entity-valued attributes, and are used by the RelationSubset operation to process a query when an attribute value list contains an entity-valued attribute.

*Indices* are B-Trees with variable-length string keys indexing TID values. More than one index entry may be present with the same key. Indices are maintained by the Model level for the names of entities in a domain, using the TID to reference the record representing the entity. The B-Tree indices on names are updated by the ChangeName and DeclareEntity operations, and may be used by the DomainSubset operation. Indices are also maintained as requested by the client for relations; these are updated by the SetF operation, and are used in the implementation of the RelationSubset operation. We describe DomainSubset and RelationSubset in Section 6.8.

### 6.2 Entities and relationships

It is worth noting that Cypress application programs reference database items via *handles* rather than the *cursors* provided by most database systems. A handle is returned by Cypress whenever an application program retrieves a relationship or entity from a database. Application programs may have any number of entity-valued or relationship-valued variables, each containing a handle that uniquely identifies the database object it currently references. Each call to NextRelship, GetF, etc., produces another such handle. In other database systems (e.g., INGRES or System R), references to tuples are made through cursors, with a higher overhead than handles. Client programs may copy whole blocks of data from a client buffer into the database tuple that is at the current cursor position, or vice versa.

We do not claim that handles are any better than cursors; we emphasize the difference because it effects the design of Cypress. Handles have the advantage of tighter coupling between the programming language and DBMS: database items simply appear as values in the programming language. Handles have the disadvantage of requiring garbage collection to determine which database references are no longer in use. The client indicates this explicitly in a cursor-based DBMS.

As already discussed, the Model level represents both entities and relationships as records. The records may be of three internal types: virtual, surrogate, or stored. Virtual records are system schema entities, e.g., DomainDomain or StringType. Stored records are schema or data records;

these are the Storage level records we described in Section 6.1. They may represent relations, attributes, domains, relationships, or entities. Surrogate records are used in conjunction with the *surrogate relation* optimization, which we describe in Section 6.6.

Note that a TID can uniquely define an entity, just as a [segment name, domain name, entity name] triple does. However, TIDs are used only *inside* Cypress and are never provided to clients. The Entities (and Relships) returned to the client are pointers to records that *contain* the TID and other information; the client may not access these data, and the database system keeps a pointer to this record so that the record may later be modified.

Keeping TIDs a "secret" in this way has a number of advantages. Note that we can find all references to records in running programs, as we keep a list of the record handles in use by clients of the system. We can find all references to records within databases, as the group lists and indices can quickly be traversed to find references to a record, and we do not store TIDs outside of groups and indices. As a result, we can:

1. invalidate references to a record when it is deleted,

2. move a record and update all references to it (we do not yet do this), and

3. re-use TIDs formerly belonging to deleted records.

Among other things, these features allow us to get by with TIDs as short as 32 bits, as we need only as many TIDs as there are records in the largest possible database. (Actually slightly more TIDs, since there is internal fragmentation due to the fixed maximum number of records per page).

We never use TIDs for references across segment boundaries, although they could in theory be used for such as TIDs contain a segment number. The lack of cross-segment references makes segments independent, so that one segment can be destroyed or rebuilt without affecting others. It also avoids the need for a common network database that maps segments to database address space. Destruction or inaccessibility of such a common database would make *all* segments inaccessible.

### 6.3 Values

The choice was made to represent values as REF ANYs in Cedar. A REF ANY is a pointer to an object whose type is determined at run-time, via a type code stored with the object. REF ANYs have some inconveniences and inefficiencies, particularly the overhead of using a pointer when it is not required (e.g., with INTs) and the overhead of the run-time NARROW operation which coerces the REF ANY to a particular type. However REF ANY seems the best solution since convenience and simplicity of the interface is important.

Because only a few of the many datatypes made available to alpha clients of the original Cedar DBMS were widely used, we implement only REF INT, REF BOOL, REF GreenwichMeanTime, ROPE, and Entity values. Some other types, e.g., pointers to Mesa records, are still used by internal procedures but are not currently exercised by client programs.

Note that in our system particular Relship or Entity variables all have the same Cedar type; i.e., there is not a different programming language type for each domain or relation. Run-time operations are provided to fetch the domain or relation to which an entity or relationship belongs. In the associative programming language LEAP (Feldman and Rovner [1969]), this run-time typing is taken to the extreme of treating a relationship's relation not as its type, but as simply another field of the record: relationships are not typed. If this approach were taken in our implementation, for example, RelationSubset would not need the first (relation) argument: the user could specify the desired relation as part of the AttributeValueList or omit it. The typeless relationships would provide little or no additional utility in our system, however, and in fact may complicate the world for a typical application. The cost of complete typelessness does not seem to justify any utility it may have.

The run-time typing of relationships, entities, and values means that a fair amount of the code in the dozen Model level operations is concerned with error checking. If at the Query level we introduce a compiler for database operations, we can save execution time by performing many of these type checks at compile-time. Fortunately the caching of schema information avoids database accesses for most of this checking, so the current overhead is not overwhelming.

The Storage level implements strings of arbitrary and variable length. The length hint supplied to DeclareAttribute is used to decide how much space to allocate in the record itself, but the characters are stored elsewhere if the field overflows. It is currently not efficient to store strings with thousands of characters in Cypress if the strings are frequently updated, because the storage allocation function is not designed to deal efficiently with data chunks larger than a page or so. We are implementing a supplementary package to Cypress which provides efficient storage of these large text strings, such as the body of a document or program. The actual text is stored in conventional files; the file, position, and size of a text string is stored in the database where it is referenced. The supplementary package also implements a *log* of all database updates so that a database client may replay or back up database actions, independent of the database transaction mechanism.

### 6.4 Caching the data schema

In order to improve the performance of the database system, we cache information about the client's data schema in a quickly-accessible form. On the first reference to an attribute, for example, the system saves the type of value the attribute demands, its relation, and its uniqueness. On subsequent uses the type-checking performed by GetF and SetF can be performed without access to the data

schema stored in the database. The attribute cache information is flushed whenever an AbortTransaction or DestroyRelation call is invoked.

Schema caches are also maintained for commonly-accessed information about domains and relations (subtypes of a domain, indices on a relation).

The schema cache is distinct from the Segment level's LRU cache of data pages. Although the schema is cached just as any other data at the Segment level, we wish to avoid all database operation overhead, for examining schema entities. The schema-caching optimization makes a considerable improvement in Cypress performance: it gives a factor of ten speedup in GetF in nearly all database applications.

Since the Storage level does not know how to deal with system records such as the DomainDomain or IntType, the Model Level must make exceptions for these entities in the implementation of a number of operations. There are currently a few limitations in the way this is done; for example the DomainDomain is not actually in the enumeration of DomainSubset[DomainDomain], and user-defined Relships may not reference system entities. GetDomainRefAttributes and DomainSubset on the DomainDomain and RelationDomain had to be coded specially, and so on. Since Datatypes (which are system entities, not stored in the database) cannot be stored by the underlying storage level, integers are stored corresponding to IntType, StringType, and so on, and these are mapped to the system entities when passed to or from the client program.

The data schema is stored in four underlying storage-level recordtypes. These recordtypes are the domain recordtype, relation recordtype, attribute recordtype, and index recordtype. The domain recordtype has as attributes the domain name and the recordtype handle used by the underlying storage level. The relation recordtype has the same two attributes, plus a special attribute which indicates whether the surrogate relation optimization has been performed on the relation. The attribute recordtype has several attributes, indicating each attribute's type, uniqueness, and relation, and, if applicable, their length and the index they participate in. Indices are represented in an index recordtype.

There are some ordering constraints on schema definition, as noted in Section 3. All of these constraints could be removed by enough special-case code, including deletion of attributes. We chose to pass the buck to the data schema editing tool described in Section 7.2 for now.

### 6.5 Links and co-location

As discussed in Section 6.1, relationships with entity-valued attributes declared with link = Linked are inserted in a doubly-linked list (a group) with the entity they reference at the head of the list. Thus

groups provide an efficient way to find all references to an entity via a particular attribute. There are three other possible values for the link hint: **Unlinked**, **Colocated**, and **Remote**.

**Colocated** is the same as **Linked**, but an attempt is made to store the relationship which references an entity on the same file page as the entity that it references. It provides even faster access to an entity from a relationship or vice versa. Only one attribute of a relation can be declared colocated. Relationships which reference an entity in this attribute are colocated with the entity and with each other.

Colocation must be performed by **DeclareRelship** when it creates a new relationship for which a value has been specified for the colocated attribute. (**SetP** calls **DeclareRelship**, so it achieves the same result; but **SetF** causes no colocation, as the record has already been allocated.) When **DeclareRelship** is called to create a relationship with a colocated attribute **a** with value **e**:

1. **DeclareRelship** attempts to store the new relationship on the same page where **e** is stored;

2. if that page is full, **DeclareRelship** attempts to store the new relationship on the same page as the most recently created relationship referencing **e** in **a**;

3. if that page is also full, or there are no other relationships referencing **e** in **a**, then the relationship is stored anywhere.

One can imagine a number of variations on this algorithm; we have not experimented with them at the time of this writing. One might profitably reserve pages for relationships referencing an entity on the basis of an estimate of the space that will be required.

NOTE: Colocation is not yet implemented at the time of this writing. Relationships are simply placed on pages in the order they are created. We currently achieve some approximation to colocation by dumping a segment in textual form, with all relationships that reference an entity grouped together, and then reconstructing the segment from the dump file.

The string name of referenced entities are stored in **Unlinked** attributes, rather than using a group list. Thus, groups cannot be used to find an entity from an unlinked reference to it, nor vice versa. When an attribute is declared **Unlinked**, the database client typically declares an index on the attribute with **DeclareIndex**, thereby making access to a relationship reasonably fast (logarithmic in the size of the relation instead of a single page access, but rarely more than a few page accesses). In other cases, an index is better than groups. Indices could use less space if entity names are short: groups require three 32-bit TIDs per record. If TIDs are simply being compared for equality, e.g., for a relational join, the TIDs may be compared directly in B-tree pages rather than fetching the pages containing the records themselves.

Note that when **GetF** is performed on an unlinked attribute, a B-Tree lookup must also be performed, to map the entity name onto the entity record that is returned. If **GetFS** is used, this lookup is bypassed.

**Remote** indicates that an attribute may reference entities in other segments (in a different segment than the relation). **Remote** attributes are currently treated much like **Unlinked**: the name is stored instead of a physical link (remember we do not want links, which are based on TIDs, to cross segment boundaries). Unlike **Unlinked** attributes, the segment and domain name must be stored in the reference as well as the entity name. The **Remote** attribute link hint is not technically necessary, as the Model level could determine how to store an entity-valued attribute "on the fly." It greatly simplifies the implementation, however.

If an index is *not* declared on an unlinked or remote attribute, the **RelationSubset** operation will take time linear in the size of the relation to process a query whose attribute value list asks for a particular entity value for the attribute. An unlinked and unindexed entity-valued attribute would be useful only if the relation is known to be of a small fixed size, or queries are never performed on the attribute.

A time comparison of three simple cases of entity-valued attributes follows: an attribute declared linked but not indexed, an attribute declared indexed but not linked, and an attribute that is neither linked nor indexed. We show the times for the three common operations of fetching the attribute, setting the attribute, and doing a **DeclareRelship** operation with the **OldOnly** option to find the relationship with a particular value for the attribute (this is equivalent to a **RelationSubset** and **NextRelship**). These and all other measurements were performed on the Xerox Dorado computer; see Section 8.2 for details.

| **Operation** | **linked** | **indexed** | **unlinked, unindexed** |
|---|---|---|---|
| GetF | 1 ms | 4 ms | 1 ms + N*.5 ms, N = domain size |
| SetF | 7 ms | 40 ms | 4 ms |
| FetchRelship | 4 ms | 32 ms | about N*.5 ms, N=domain size |

These measurements were performed on relationships whose pages were already in the Segment level's cache, thus the "linked" column is near the ideal time one would obtain if records were colocated. The difference between linked and colocated times depends on a particular application's data schema, of course. A page read and write both require approximately 60 ms. Thus **GetF** and **SetF** could take as much as 60 ms in the worst case. For typical applications, the payoff from colocation is much smaller than that, but still seems to produce more than a factor of two speedup in both **GetF** and **SetF**.

If the surrogate relation optimization is applicable, the basic operations are still faster. GetF on a linked entity-valued attribute, for example, averages about 0.3 ms. This compares to 1 ms in the above table.

Note that the introduction of entities to the relational model allows the DBMS to make some intelligent guesses as to where to use colocation and links. By default, an attempt is made to link and colocate relationships with the entities they reference. This may be overridden by the client, of course.

Our current database applications most frequently retrieve together all or most of the relationships, from a variety of relations, which reference a particular entity. Most implementations of the Relational model store the records of a relation contiguously. The relational arrangement results in many more page reads than ours for the entity-centric access we find most common.

### 6.6 Surrogate relations

The surrogate relation optimization is used when DeclareAttribute is called to define the first attribute of a relation, and: (1) the attribute is entity-valued, (2) it is a primary key of its relation, (3) its type is a domain with no sub-types, and (4) that domain does not yet have any entities defined in it. The relation is then specially tagged as being a surrogate relation, and the 2nd through Nth attributes will be added to the *domain's* recordtype, i.e., the data will be stored in the entities rather than separate records. The relation itself will not exist as a recordtype, and any relationships in the relation will be represented as surrogate records. Thus the records that represent entities may have fields in addition to the entity name, containing surrogate relation attributes.

The data schema must record the relations for which the surrogate relation optimization has been performed, for internal use. This information can also be useful to select clients such as the data schema update tool.

When compared to colocation (i.e., first attribute declared Colocated), the surrogate relation optimization is an improvement in two ways. First, a surrogate relationship uses less space (no additional record and link overhead); this may allow information to be stored on the same page. The overhead of an additional page access (60 ms) can make a considerable difference in an operation that takes less than 1 ms (GetF). Also, accessing the field directly rather than following the link to another record is somewhat (currently about .4 ms) faster even when the other record is on the same page.

The effect of the surrogate relation optimization on Cypress performance is of course dependent upon its applicability to a particular application's data schema. In general, we find that the surrogate optimization can be performed on about half the relations in the schema. For the electronic mail

database application, construction of database entries runs about 30% faster with the surrogate relation optimization enabled (instead of just colocation). Other operations, e.g., displaying a message, have equal or better performance improvement.

We could allow database clients to specifically indicate when the surrogate relation optimization should be performed, as we do with links. The optimization appears desirable whenever possible, however, so this unnecessary.

Note the restrictions mentioned earlier on the application of the surrogate relation. The restriction that a domain have no subtypes is made to simplify the problem of assigning field positions in the recordtypes. The restriction that the domain be empty is made to avoid reconstructing or marking existing entities which would not have the field. The latter restriction is similar to one on relations, that attributes may only be added to a relation while it is empty. The schema editing tool described in Section 7 makes such restrictions invisible by copying the domain or relation when necessary.

### 6.7 Basic operations

In this subsection, we discuss more details of the basic operations on entities and relationships.

DeclareEntity and DeclareRelship simply create a record in the underlying domain or relation recordtype, updating any associated index entries. The DeclareEntity operation uses the DomainSubset operation to determine whether an entity with the given name already exists. DeclareRelship similarly uses RelationSubset to determine whether the given relationship already exists, unless version is NewOnly.

A small complication is introduced by the interaction of the surrogate relation optimization and the use of indices. The attributes of relationships of a surrogate relation are stored as "attributes" of entities in the target domain. Index entries reference the entities. This would suggest that we must create index entries when the *entities* are created. The semantics of null values can be used to improve efficiency, here: we need make no entries until values are actually assigned. This should not be done until surrogate relationships are created and their attributes assigned values.

GetF and SetF must make special checks for the surrogate relation optimization; when this optimization has been performed, the storage level "field handles" for attributes of the relation actually reference fields of the domain, in which the relation's attributes are stored. With SetF, a particular constraint has currently been added: the key attribute of a record must be assigned before others, so that the Model level knows which entity in the domain is being referenced. Note this would not be a problem if we disallowed SetF, requiring that whole relationships be created or destroyed as units.

Type checking is necessary only for one operation, SetF. If the domain sub-type hierarchy is not used, i.e., an entity stored in an entity-valued field is precisely of the domain required by the attribute, the check is fast. Furthermore, if type-checking is thwarted altogether, i.e., the attribute is of type AnyDomainType, the check is even faster: the system need only check that the base type (i.e., entity-valued, string-valued, etc.) is correct. In the case that a check on the attribute's domain type fails, SetF recursively searches *up* the hierarchy from the actual entity type looking for the attribute's type. It fails upon finding no more super-types. (Searching upward is faster than searching downward for randomly-selected examples in the domain type lattice in the largest application schema we have so far.) More efficient schemes could be implemented if necessary, e.g., pre-computing the super-domains and redundantly storing the list with each domain.

SetF must update group lists when they are being maintained for entity-valued attributes, as described earlier. Conversely, if links are *not* maintained for an entity-valued attribute, GetF must perform an index lookup to map the stored entity name into the entity handle returned to the client (GetFS need not do this lookup, however, as it is only required to return the name). SetF uses RelationSubset to determine whether key constraints are violated by the new attribute value.

The algorithm for DestroyRelship is relatively simple. The Storage level operation to destroy the record automatically deletes the record itself and removes it from the linked list maintained through all records of its recordtype. Variable-length strings which have overflowed the space allocated in the record (the length argument to DeclareAttribute) are stored in separate pseudo-records that must also be de-allocated. Also, any index entries for the destroyed Relship must be deleted, and the relationship must be removed from any group lists in which it participates.

DestroyEntity is similar to DestroyRelship, but any relationships that reference the entity must also be destroyed. There are three special cases of relationships referencing an entity:

1. Surrogate relationships stored in the entity record itself must be destroyed; the comments about deleting variable-length strings and updating groups and indices apply as in DestroyRelship.

2. Relationships linked to the entity via a Linked attribute may be found and deleted quickly.

3. Relationships referencing the entity via an Unlinked attribute must be found by a RelationSubset operation on attributes that can reference an entity of the kind we are deleting. A group list of these unlinked attributes is maintained in the database for each domain, to make this operation more efficient.

The Eq operation returns TRUE iff its arguments have identical TIDs. If its arguments are from different segments they will have different TIDs, of course, as the segment numbers are stored as the

high order bits of the TIDs. The correspondence between segment numbers and atoms is maintained to implement the SegmentOf operation.

The procedure GetAllRefAttributes is implemented using a special Storage level operation that returns all the groups in which an [entity] record participates. As unlinked attributes will not be found by this operation, they must be added to the list returned by GetAllRefAttributes, as well.

### 6.8 Aggregate operations

In this section, we discuss details of the DomainSubset and RelationSubset algorithms that have not already been covered.

The algorithm for DomainSubset is relatively simple. If a name or range of names is specified in the call, the index for the domain is used; otherwise the linked list the Storage level provides through all the records of the domain recordtype is used. If it is requested that subdomains be searched, a DomainSubset operation is invoked on each in sequence. Whatever the method, the information about the enumeration is packaged up in the EntitySet returned so that NextEntity can sequence through the indices, lists, or domains as required.

The algorithm for RelationSubset is somewhat more complex than DomainSubset. We are given an AttributeValueList, composed of triples of the form [attribute, low, high] constraining the given attribute to have a value greater or equal to low and less than or equal to high. The high value is typically omitted and is then assumed to equal low. The high value *must* be omitted for entity-valued attributes, since a range of values makes no sense in that case. RelationSubset must choose an efficient manner in which to find the relationships satisfying the list.

The following strategies are used by RelationSubset, in this order:

1. If the surrogate relation optimization has been performed on the relation, then (a) if a key-valued attribute is contrained to equal a particular entity in the attribute-value list passed to RelationSubset (the constraint list), we may return a single-element RelshipSet; (b) otherwise, we set a boolean in the RelshipSet (the one we will return) to indicate that whichever of the strategies 2, 3, and 4 are used below, NextRelship must convert the entities we obtain to surrogate relationships.

2. If one of the attributes in the constraint list is entity-valued, then we use the group list for the given entity value. That constraint is removed from the list, and the remaining *reduced list*, is saved to be serially checked by NextRelship against records in the group list.

3. If there is an index on one or more of the attributes in the constraint list, the index

containing the largest subset of the attributes is used to find the relationships satisfying that subset of the list. The remaining reduced list is checked by **NextRelship**, to filter the relationships enumerated from the index.

4. If all of the above strategies fail, the underlying storage-level operation to enumerate the list through all records of the relation's recordtype is invoked, and **NextRelship** serially filters out the relationships satisfying the **constraint** list.

The **NextRelship** operation enumerates the appropriate group, index, or recordtype as specified by the **RelshipSet** that **RelationSubset** produced. As all three of these enumerations are bi-directional at the Storage level, **PrevRelship** is a straightforward inverse of **NextRelship**. Note the slight complication in **NextRelship** and **PrevRelship** introduced by the surrogate relation optimization. In that case, the indices, group lists, or recordtypes can be scanned as usual, but refer to entities rather than the surrogate relationships which have been mapped to them. Surrogate relationships are created on-the-fly.

The time required by the **RelationSubset** operation is of course dependent upon the path through the above algorithm taken, whether an index or group is scanned, and so forth. A typical application averages about 7 ms for a variety of **RelationSubset** calls. Thanks to the schema cache, only about half of this time is spent examining the data schema to check the validity of the query and determine which physical access path to use to process it.

The time for **NextRelship** is typically 0.4 ms, and is nearly the same for a group, index, or recordtype enumeration. Currently most of this time is for the Cedar allocation of the **Relship** handle itself! However, the **NextRelship** time will be much larger for a database application without a working set in the database page cache: the time to fetch pages will dominate.

We will not discuss the **MultiRelationSubset** operation and Query level here, as they have not yet been constructed. One possible plan would be to (1) build a parser at the Query level to map text queries into a tree representation, (2) make a major extension of the **RelationSubset** operation to optimize access path selection over the entire query, (3) make a minor extension to **RelshipSets** (and **NextRelship**) to allow them to act upon nested relations (still producing a single relation at the top level), and (4) build a Query-level formatter which prints the **RelshipSet** enumeration for the client.

### 6.9 Summary

The Cypress DBMS is built in four levels: the Model, Storage, Segment, and File levels; in this section we have concentrated on the implementation of the Model level. We summarized the optimizations made in that implementation:

1. caching the data schema,
2. use of B-tree indices upon entity names and relation attributes,
3. use of "group" links through relationships referencing an entity,
4. co-locating relationships which reference the same entity on the same page, and
5. storing relationships which reference an entity as surrogates in the entity itself.

Caching the data schema is crucial to good performance; almost all operations must examine the data schema to check the legality of the client's request or to decide how to perform the operation. The payoff can be as much as 90%.

The payoff from the use of indices is variable, but they are crucial upon entity names and are often beneficial upon relations. There is a trade-off between using indices and links on relations; indices can be used on multiple attributes and use less space if the keys are short, but links are typically a factor of three faster.

It is enlightening to observe the progressively better performance for the GetF operation on an entity-valued attribute that is (1) indexed, (2) linked, (3) colocated, or (4) surrogate. As just noted, links are typically a factor of three faster than an index, as at most one page (other than the page on which the relationship resides) is accessed. Colocation typically produces another factor of three or so, though this varies widely with database application. The surrogate relation optimization provides another factor of two or so, by avoiding another tuple lookup (on the same page). The best time for a GetF is currently about .2 ms.

In Section 5, we showed how the Cypress data model was designed to provide a higher-level conceptual model, simplifying the development of database applications. In this section we have shown that a higher-level data model can also be used to improve the performance of a database system, by capitalizing on the additional knowledge of data semantics and interconnectivity. In Section 7 we will concentrate on the third advantage of a higher-level model: the more powerful tools and integration of database applications it enables.

# 7. Database environment and applications

## 7.1 Database environment

Along with the development of the Cypress DBMS, adding the Model level facilities to the former Cedar DBMS, we built a system named Squirrel to provide some general-purpose database tools and some level of integration between database applications. Several applications of the database system have been developed. These applications include Walnut, a database of electronic messages, and Hickory, a calendar/appointment database.

Squirrel provides the end user with tools to perform a number of basic database operations. These tools include:

1. A browsing window to examine database entities and relationships.

2. An editing window to modify database entities and relationships.

3. A simple query window to search databases.

4. A facility to dump databases in textual form, and subsequently reload them. The external form is human-readable text, and contains the database schema as well as the data.

5. A facility for examining and modifying the database schema, automatically re-organizing existing data for simple schema changes.

These five basic functions may provide all a user requires for simple kinds of database applications. For example, for some personal databases (a wine database, and a database of phone numbers and addresses) we have found the Squirrel tools useful in and of themselves. Squirrel is also used by database application programmers to examine and modify a database in the process of testing and debugging application programs.

In addition to the facilities directly provided the end user, Squirrel provides facilities for application programs. Squirrel's function in this regard is to encourage a consistent and integrated user interface among the various applications. The goal is a confederation of database facilities as opposed to monolithic independent programs. Squirrel provides some convenient procedures for common database operations and Cedar Viewers operations, common conventions for the user interface, and a mechanism for applications to communicate with one another when data is displayed and updated.

The user interface paradigm we use provides one Cedar Viewer on the screen per database entity, an object-oriented menu of commands upon a database entity, and a consistent interpretation of user selections on the screen. The paradigm is based upon three basic types of application windows: *displayers*, *editors*, and *queryers*. A *displayer* displays a database entity, provides a menu of commands upon it, and shows the information the database contains about the entity. For example, for an electronic message displayer, the message's header and body would be displayed, along with commands such as Answer, Forward, and Delete. An *editor* looks much like a displayer, but allows the user a form with which to modify the information about the entity, and usually has a different set of commands. For example, a message editor is created in response to a NewForm or Answer command in the Walnut system; the user fills in the fields and invokes the message editor's Send or File command. A *queryer* also looks like a displayer and editor, but the user may fill in the fields with values, boolean expressions of values, or other application-interpreted information; the queryer represents all entities in its domain which satisfy the constraints. When the user invokes the Query command, the entities satisfying it are displayed and may be browsed or printed.

Each database application program, such as Walnut or Hickory, *registers* itself with Squirrel by passing procedures to be called when an event of interest to the application occurs. The application may register a displayer, editor, or queryer procedure to be called when the user requests one for a particular domain. Walnut registers such for the message domain, for example. An application may also register itself to be called when certain database relations are updated, so that it may update its display.

Figure 7-1 illustrates a variety of types of displayers. Note how each application takes best advantage of the display for the particular type of entity involved. Figure 7-1a is a message displayer, implemented by Walnut. Figure 7-1b shows a month displayer, implemented by Hickory. Figure 7-1c shows a picture displayer, implemented by a simple image application. Figure 7-1d shows a relation displayer, implemented by a data schema display package. Figure 7-1e shows a "whiteboard" displayer, used for spacially organizing and accessing entities. Figure 7-1f shows a "default" displayer which is invoked when no particular application deals with the given domain (in this case, the Organization domain). Squirrel implements the default displayers, whiteboard displayers, and schema displayers.

Figure 7-2 contrasts a displayer, editor, and queryer for messages.

Figure 7-3 illustrates one way in which the collection of database applications may appear as an integrated collection of facilities to the user, showing successive windows on the screen as the user browses from a "Whiteboard" display, to a "Message Set" on that whiteboard, to a "Message" in the set, to the sender of the message. The user crosses application boundaries in going from one kind of entity to another, but the action to browse (pointing with the mouse and pushing its center button) is the same throughout.

Squirrel, Walnut, and Hickory will be more fully described in a future report. To put the database work in context, however, we will summarize Squirrel in Section 7.2 and the applications in Section 7.3.

## 7.2 Database tools

A Squirrel window on the screen provides the user with the basic database functions, such as commiting or aborting a transaction, dumping or loading a database, or erasing portions of a database. The Squirrel window also allows the user to explicitly invoke a displayer, editor, or queryer on a particular domain or entity. If no application has registered itself for the given domain, Squirrel invokes its default application-independent displayer, editor, or queryer when the user requests one.

The default entity displayer shows the domain and name of the entity displayed at the top of a Cedar Viewer window, and all the relationships in the database which reference that entity are shown as the main body of the window. An example of such a window is in Figure 7-3d. The relationships are displayed in the form

**relation attribute1: value1 attribute2: value2 ... attributeN: valueN**

where **relation** is the relationship's relation and the **attribute: value** pairs specify the values of its attributes. The attribute which references the displayer's entity is not shown, as it is normally redundant (all of the relationships in the displayer for an entity have at least one attribute which reference the entity or they would not be displayed). The other attributes are displayed in the obvious way for string, integer, and boolean values. For entity values, the name of the entity referenced is displayed. In addition, for entity values, the user may select the entity with the center (yellow) button on the mouse, causing a displayer to be created on the screen for the selected entity. This selection with the yellow-button provides the basis for browsing, and is supported by Squirrel displayers as well as application displayers.

The default editor window also shows the domain and name of the entity being edited at the top of the window. However it differs from the default displayer in that an editable *form* is provided in the body of the window for the entity, showing not only those relationships which already reference the entity, but "blank" relationships for relations which could potentially reference an entity of its domain. A "blank" relationship appears as a relationship in a displayer, but the values are given as blank fields that can be filled in by the user by selecting the blank with the red mouse button and typing. If the editor window is on a new entity (one that previously did not exist in the domain), then all the relationships shown will be blank ones. Special combinations of mouse buttons invoke special commands on the blank relationships: creating a new blank relationship for a particular

relation, explanding the blank relationship to show the types expected of the fields, or specifying whether the system should automatically create an entity when a field value is filled in with an entity name that does not exist in the appropriate domain. Menu buttons at the top of the editor can be used to change the name of an entity or merge it with another entity.

The default queryer window has a form similar to an editor window, except that only a domain name is shown at the top of the window. The queryer window, unlike an editor window, represents any number of entities in a domain which satisfy a query. The query is specified by filling in the form fields with expressions that specify the desired values of the fields. The expression may include boolean ORs (written as "|"), and also ranges of values (written using "~", e.g., "5~11"). After the form has been filled in and the user uses the window's Query command, the queryer opens a new window on the screen displaying the names of the entities which satisfy the query. The client may scroll through the entities, or select one of them with the yellow mouse button to open a displayer on it. Note that except for the form in which the answer is examined, this queryer window is similar to the Query-by-Example system of Zloof [1975].

In addition to displayers and editors on ordinary data items, Squirrel implements special displayers and editors for domains and for relations. The displayer for a domain shows the attributes in the data schema which can reference an entity from that domain, the sub-types and super-types of that domain, and the entities of the domain sorted by name. The user can scroll or yellow-select the entities. The editor for a domain shows only the sub-types and super-types, and allows the user to change these. The editor will automatically copy entities and rename domains if necessary to achieve the effect of a requested change (because the underlying database system does not allow defining sub-types of a domain that already contains entities). The displayer for a relation shows a table whose columns are labelled with the names, types, and uniqueness constraints of the relation's attributes. An example of a relation displayer is shown in Figure 7-1d. The rows of the table show all of the relationships in the relation. Again, the user may scroll through the relationships, and may yellow-select attribute values that are entities to open a displayer window. The editor for a relation shows only the attribute names, types, and uniquenesses, and allows the user to change these. The user may also delete attributes of the relation or create new ones; the relation editor will automatically copy all the relationships into a new relation with the same name, since the underlying database system does not allow changing attributes of a relation after relationships exist.

Squirrel and application programs support a common interpretation of mouse buttons for the user. This common interpretation is as follows:

| Button | Interpretation |
|---|---|
| Yellow | Open a new displayer on the entity at the current position |
| Red | Select the entity or datum value at the current cursor position |
| Blue | Extend a selection made by red to the current position |
| Ctrl-Red | Delete the relationship at the cursor position |
| Ctrl-Blue | Extends the action of Ctrl-Red over a number of relationships |
| Shift-Red | Copy the entity or datum value at the cursor into the current input focus |
| Shift-Yellow | Expand the selected entity in place (show more information) |
| Shift-Blue | Extend selection with Shift-Red |
| Others | Currently application-defined |

The yellow button was dedicated to the "show me" interpretation because this was deemed the most common operation upon a selection, to open up a new window on an entity. The user does not always want to open a *new* window, however; this tends to proliferate windows on the screen. The Squirrel windows and database applications currently follow the following convention for opening new windows when a user yellow-selects an entity:

1. If the user has yellow-selected an entity in this window before, and the new displayer window thus created is still on the screen, we *re-use* that window, replacing the entity that used to be displayed in that window with the selected one.

2. Otherwise, we create a new entity displayer on the screen for the selected entity, in non-iconic form on the left-hand side of the user's display screen.

The interpretation of the mouse buttons was chosen to be as compatible as possible with existing interpretations in the Cedar Viewers environment. There is in fact some overlap. For example, one can select an entity by selecting the entire window (or icon, if in iconic form) for its displayer. It can then be used by a command, e.g., to add it to a set of entities in another window.

All of the operations afforded the user by Squirrel are also available to client programs, through a Cedar Mesa interface called "Nut." The Nut interace is also the one through which application programs register themselves with Squirrel, through the Register operation:

```
Register: PROC[
    domain: ROPE,
    display: DisplayProc← NIL, -- to make displayer for entity in domain
    edit: EditProc← NIL,  -- to make editor for new or old entity in domain
    query: QueryProc← NIL, -- to make queryer on domain
    create: CreateProc← NIL, -- to create viewer for above (defaults to container)
    update: UpdateProc← NIL, -- to call when database is updated
    ];
```

### 7.3 Database applications

Our first application of the Cypress DBMS was Walnut, a collection of facilities for sending, receiving, filing, and querying electronic messages. Walnut provides four kinds of Cedar Viewers visible to the user:

1. *Control window*: This viewer window is the one that appears when Walnut is started. It provides menu commands to fetch new mail, create a new message form (item 4 below) and create a new "message set" (item 2 below). The control window also reports when new mail is available, or an error is recognized by the Walnut system.

2. *Message-set displayer*: This window displays a list of messages, their senders, dates, and subjects. There is a special "Active" message set into which new mail is normally inserted. The viewer for this message set is automatically put on the screen, along with the Walnut control window, when Walnut is started. The user may display many message sets on the screen at the same time. Yellow-selecting one of the messages produces an instance of item 3 below, a message displayer. A menu of commands at the top allows removing messages from the message set and/or adding them to other message sets.

3. *Message displayer:* This window displays a message in the database. The menu of commands at the top includes Answer and Forward, that produce an appropriately initialized instance of a message form (item 4 below), and Erase, which deletes the message itself and removes it from all message sets. The user may yellow-select a message field to see a database item named there. For example, selecting one of the "Categories" items causes the corresponding message sets to be displayed (which enumerate messages on that subject or in that message set, which can in turn be selected). Selecting persons in the "To," "From," and "cc" fields causes the person to be displayed (phone number, office number, picture, or whatever the database happens to contain).

4. *Message editor:* This window displays a message form which the user can edit. Some of the fields will be initialized according to whether the message editor window was created by the "Answer," "Forward," or "New Form" commands.

5. *Message queryer:* This window also displays a message form which the user can edit. It represents all the entities with the values the user specifies in the fields he fills in. Value ranges may be specified. For example, the user might ask for all the messages with sender "Cattell" from last week, containing the string "Cypress" somewhere in the message body. When the user invokes the Get query command on the queryer, a new message set displayer appears on the screen containing the set of messages which satisfied the query. Although the messages in this message set are not actually stored in a database message set, the user may select and browse through messages in the window just as with any other message set.

Walnut uses a special text logging package, for two purposes. The current Cypress implementation does not deal efficiently with large string-valued attributes that are frequently created and deleted (the storage allocator is oriented towards objects less than a page in size). The text logging package allows Walnut to store the bodies of messages in a file, storing a pointer to the text body as a relationship connected to the message entity in the database. The text logging package is also used to maintain a record of every database-updating operation performed by Walnut, e.g., receiving a new message or adding a message to a message set. The log may be replayed to reconstruct a database after a crash, or in the unlikely event that an entire segment is destroyed. The ability to replay the log makes it unnecessary for Walnut to commit a database transaction after every operation. The text logging package used by Walnut is now being rewritten to incorporate it as an extension to the common database software, making it useful to all database applications.

Another database application now in preliminary use is Hickory, a Calendar/Clock database application for recording appointments and reminding the user of them. Hickory implements "event," "event-set," "day," "month," and "year" entity displayers. Hickory makes use of the database system hierarchy of domains to define a number of types of events, such as meetings, seminars, and trips. The user may enter particular events, or sets of events such as periodic events (e.g., a seminar that Hickory will automatically enter every Tuesday). When the client views a particular day or month he can see the events scheduled for that day or month.

A third application, "Whiteboards," has actually been integrated into Squirrel itself, because the functionality it provides is useful in conjunction with all applications. The Whiteboard package provides a mechanism to lay entities out in a tree-structured, spatial dimension. Consider three ways of moving around in a database by yellow-selection:

1. Browsing in the network by selecting adjacent entities, e.g., selecting the sender of a message to get a displayer on that person, or selecting some day in a month to get a displayer on that day. This basic browsing functionality is provided by convention in Squirrel and all applications. Let's call this "ground" level browsing (Ground Squirrel).

2. Browsing in the network by moving up and down a hierarchy of whiteboards whose ultimate leaf nodes are non-whiteboard entities. This adds a vertical dimension to (1), and is currently provided by the addition of Whiteboards to Squirrel. We might call this "tree" level browsing: although strictly the hierarchy could be a directed acyclic graph, it's easier to understand as trees (Tree Squirrel).

3. Browsing in the network by moving in a two dimensional continuum rather than the discrete vertical "chunks" provided by (2), as in Herot [1981]. All the database entities would be laid out in a two-dimensional plane, probably automatically according to their "closeness" (= the number or importance of the relationships between them). Zooming

would be continuous rather than discrete as in (2). Let's call this kind of browsing "flying:" unlike climbing in a tree, there are no discrete horizontal or vertical levels (Flying Squirrel).

Only the first two kinds of browsing have been implemented, although some experimentation with the third kind might prove it valuable.

In addition to the applications already described, several database applications are under consideration for future development:

1. *Whitepages.* A database of people, organizations, phone numbers, addresses, and their photos. Note there are both public and private versions of whitepages, and the user would like to see both public and private information superimposed.

2. *Program database.* A database of Cedar Mesa program modules and interfaces, in which the user could query the locations of definitions and uses of procedures, types, and variables. This application would be interfaced with the program editor and compiler to automatically maintain the consistency of the database.

3. *Super notebook.* A database of one-sentence to one-page ideas, interconnected and indexed in a database of logical dependencies, bibliographic references, project notes, etc. Again, this could be both a private notebook or a shared notebook.

### 7.4 Summary

We have described some tools and applications built upon the Cypress DBMS. Applications have been in regular use for over a year, and have guided as well as demonstrated the utility of the Cypress data model and implementation. In the design of database applications, we have tried to achieve some uniformity so as to present the user with a confederation of information management functions rather than independent programs.

The reader may see how the data model has influenced Cypress applications and tools. The introduction of entities to the data model enabled the object-based user interface paradigm. The information about the types of relationship attributes and their relation keys is used in the default editor to check input and automatically create entities where required. The presence of entity names and the distinction between entities and relationships greatly simplifies dumping and reloading databases in a linear and human-readable form.

Figure 7-1. Each application implements displayer windows for one or more types of database entities. Note that the type (domain) and name of the entity displayed is shown in the black bar at the top of each window, followed by a menu area in which commands appropriate to the particular type of entity are shown, followed by a display of information about the entity (tailored to the type).

**Msg: cattell.pa $ 3#170@29-Nov-82 12:09:48 PST**

Answer Forward DeleteMsg AddTo RemoveFrom DBText Freeze

Date: 29-Nov-82 12:09:55 PST
From: cattell.pa
Subject: Re: Tioga/Walnut Suggestion
In-reply-to: Horning's message of 24-Nov-82 15:42:24 PST
To: Horning.pa
cc: TiogaImplementors↑, WalnutSupport↑, Donahue

Jim,

My motto for this year is integration of applications, so I'm all for your philosophy. But the kind of "integration" of having Tioga have a button to convert to WalnutSend windows and vi versa sounds like it will ultimately end up in spaghetti! I fear we will I end up with a button convert an X into a Y window for all N squared applications.

I don't think I'm objecting just on the grounds that its hard to implement underneath with the mutual compilation interdependencies and converting one kind of viewer into another. Rather think the integration between applications should be achieved wherever possible by adding the smarts to a common controll system instead of having everyone know about everyone else. (I the sense that Viewers is a common control system for display management, Squirrel for the database applications).

Figure 7-1a. A message displayer, implemented by the electronic mail application.

**December, 1982**

Freeze Save Reset AddSelected Year Month Week Day Search Level Commit

| | | | | |
|---|---|---|---|---|
| Holiday | MDGReminder | MDG | Hickory History | Referee reports due |
| Forum | Cypress Discussion | Dealer | NewSet | |

Nov 29, 82 -- Dec 5, 82

Dec 6, 82 -- Dec 12, 82

Dec 13, 82 -- Dec 19, 82

December 13, 1982

December 14, 1982

December 15, 1982

December 16, 1982

Continuing Events: None

| Morning | Afternoon |
|---|---|
| 11:00 Event: MDG | 4:00 PM Event: Forum |

December 17, 1982

December 18, 1982

December 19, 1982

Figure 7-1b. A month displayer, implemented by the calendar application.

Picture: PARC.ais

Freeze

Figure 7-1c. A picture displayer, illustrating graphic flexibility of the window display.

Relation: member

Edit Freeze

member:

| of | is | as | in |
|---|---|---|---|
| Organization (NonKey | Person (NonKey) | StringType (NonKey) | StringType (NonKe |
| FPA | Barrows, Lillie | | |
| HP CS Lab | Raphael, Bert | manager | |
| IBM Development San | Kent, Bill | | |
| Phone List | Chetkovich, Carol | | |
| Phone List | Cattell, Eric | | |
| Xerox PARC CSL | Bogener, Janet | secretary | |
| Xerox PARC CSL | Nix, Bob | student intern | 1981 |
| HP Distributed Informa | Beech, David | | |
| HP Distributed Informa | Feldman, Samuel | | |
| RTI Technology | Rowe, Larry | co-founder | |
| MIT CS | Gifford, Dave | | |
| HP Labs | Doyle, John | manager | |
| Computer Research Ce | Birnbaum, Joel | manager | |
| Voice Project | Stewart, Larry | | |
| HP Distributed Informa | Fraley, Bob | temp manager | |
| Tandem Computers | Youssefi, Karel | | |
| Tandem Computers | Smith, Ed | | past |
| Tartan Labs | Dietz, Bill | | |
| Coherent Inc | Buechel, Linda | | |
| Phone List | Larsen, Bill | | |
| IBM Research San Jose | Krenek?, Judy | secretary | |
| Phone List | Dake, Gene | | |
| Phone List | Barnes, Joan | | |

Figure 7-1d. A relation displayer (relations are entities, too). Shows the attributes of the relation as columns, and the relationships in the relation as rows.

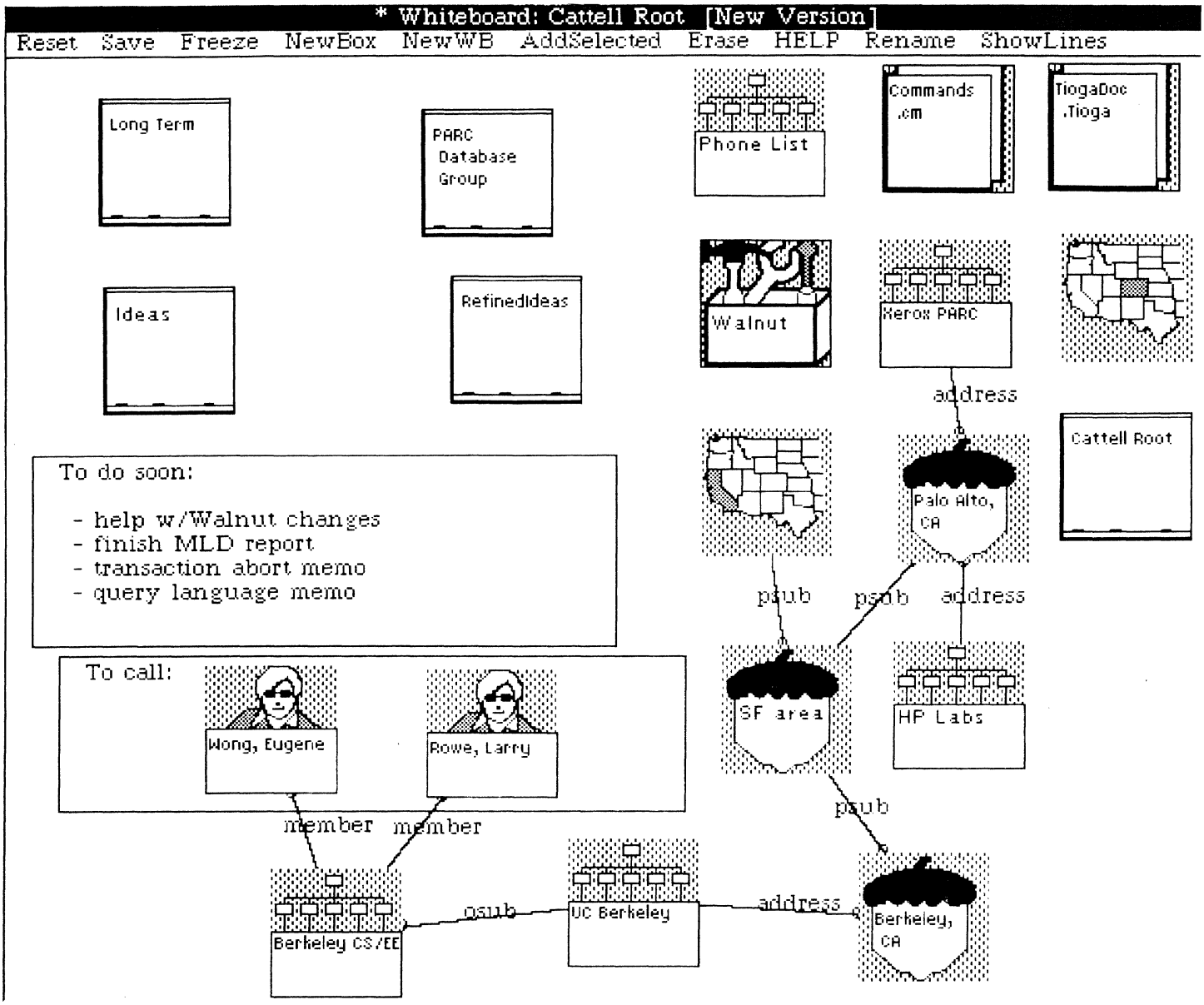



Figure 7-1e. A "whiteboard" displayer, providing a mechanism to spatially organize and browse through entities. Note that icons appropriate to the entity type (month, person, message set) are shown.

**Organization: Berkeley CS/EE**

Edit Freeze

member is: Blum, Manuel as: professor
member is: Graham, Susan as: professor
member is: Kessler, Peter as: grad student
member is: Wong, Eugene as: professor
member is: Stonebraker, Mike as: professor
member is: Henry, Robert as: grad student
member is: Katz, Randy as: grad student in: 1980
member is: Rowe, Larry as: professor
member is: Morgan, Tom as: grad student
osub of: UC Berkeley
project is: INGRES
publisher of: An Interactive Program Verifier

Figure 7-1f. A "default" displayer, invoked when no application has registered itself to deal with entities of the type (the Organization domain, in this case).

Figure 7-2. Independent of domain, three types of windows may be implemented by an application: displayer, editor, and queryer windows. Here we illustrate the three kinds of windows for the message domain.

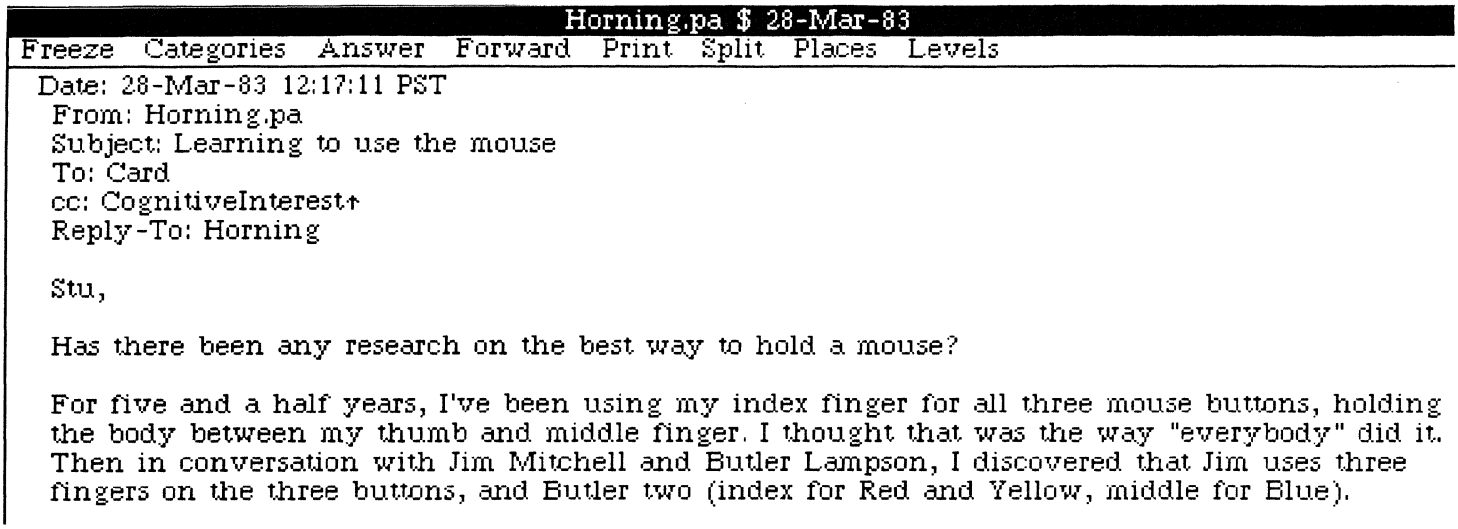

Horning.pa $ 28-Mar-83
Freeze Categories Answer Forward Print Split Places Levels
Date: 28-Mar-83 12:17:11 PST
From: Horning.pa
Subject: Learning to use the mouse
To: Card
cc: CognitiveInterest↑
Reply-To: Horning
Stu,
Has there been any research on the best way to hold a mouse?
For five and a half years, I've been using my index finger for all three mouse buttons, holding the body between my thumb and middle finger. I thought that was the way "everybody" did it. Then in conversation with Jim Mitchell and Butler Lampson, I discovered that Jim uses three fingers on the three buttons, and Butler two (index for Red and Yellow, middle for Blue).

Figure 7-2a. A message displayer, allowing message examination and browsing.

Figure 7-2b. A message editor, allowing creation of new messages to be sent or filed.

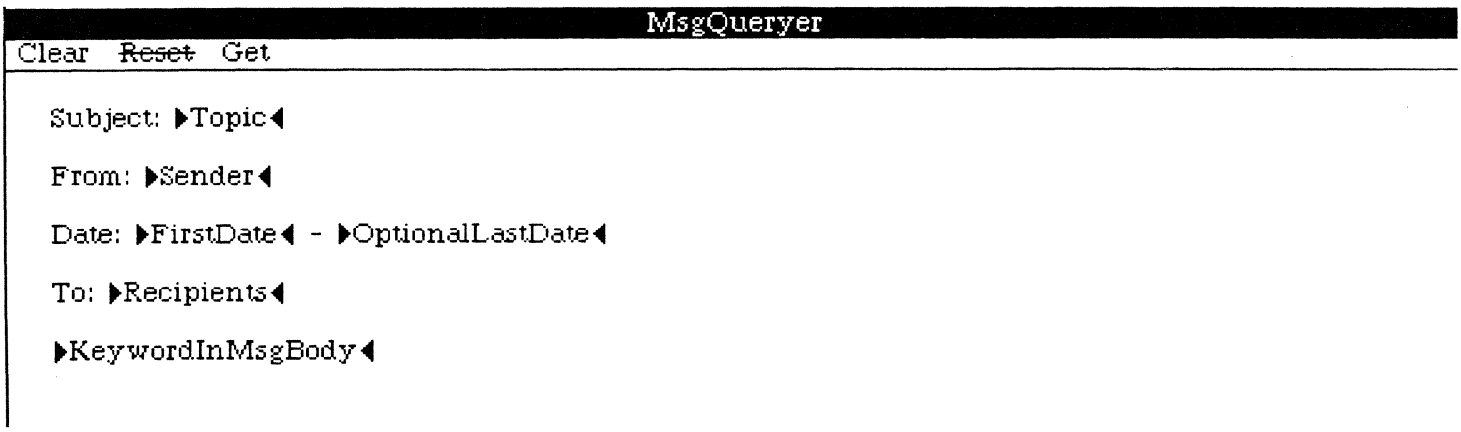



Figure 7-2c. A message queryer. User fills in properties to search the database for matching messages.

Figure 7-3. The user may browse through successive displayers by using the "mouse" pointing device to indicate entities he would like to see. In this succession of frames (Figures 7-3a through 7-3d) he has browsed from a whiteboard to a message-set it contains, to a message in that set, to the sender of the message. Each of these displayers is implemented by a different application, except for Figure 7-3d which is an instance of the default displayer.

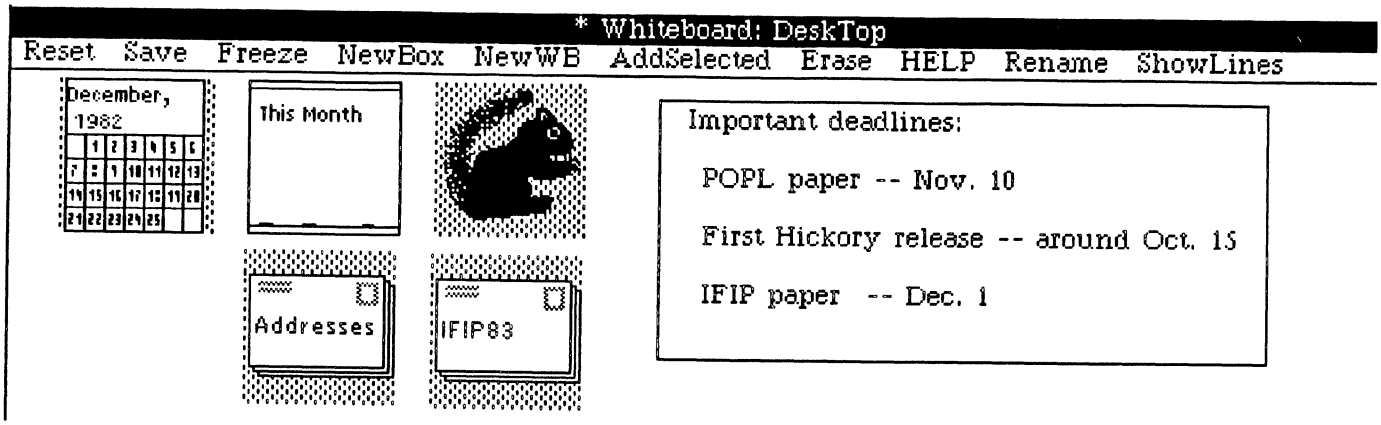



Figure 7-3a. A whiteboard displayer, showing a number of entities in iconic form.

MsgSet: Addresses
MoveToMsgSet(s) Current/Next DelFromMsgSet
11-Aug-82 Morris.PA Change your routing tables
5-Dec-82 Ed Lazows... Mail Server
5-Dec-82 Ed Lazows... More Service
25-Nov-82 cattell.pa phones & addresses

Figure 7-3b. A message set displayer, selected from the whiteboard in Figure 7-3a.

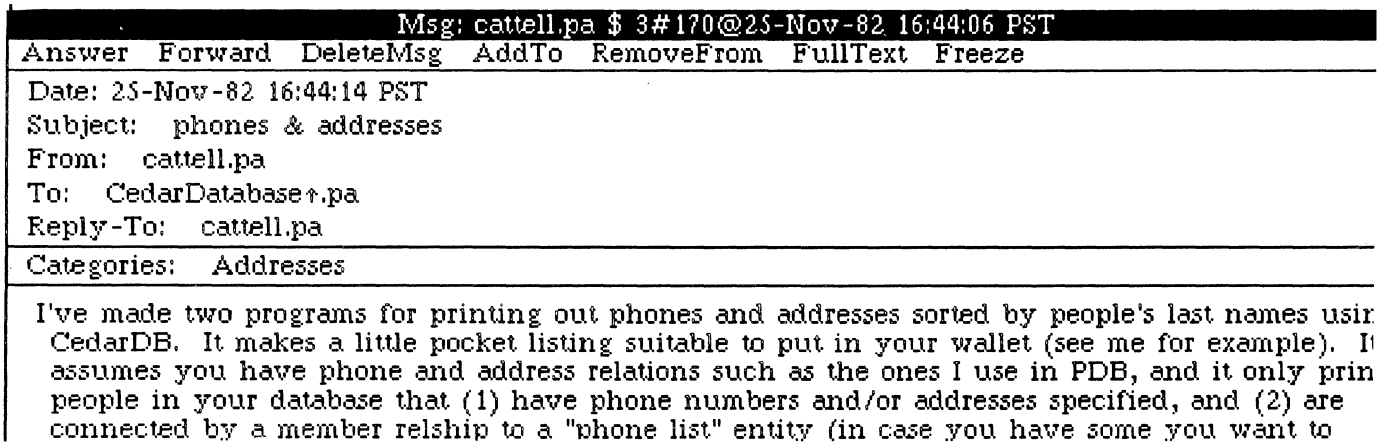

Msg: cattell.pa $ 3#170@25-Nov-82 16:44:06 PST
Answer Forward DeleteMsg AddTo RemoveFrom FullText Freeze
Date: 25-Nov-82 16:44:14 PST
Subject: phones & addresses
From: cattell.pa
To: CedarDatabase↑.pa
Reply-To: cattell.pa
Categories: Addresses
I've made two programs for printing out phones and addresses sorted by people's last names usir
CedarDB. It makes a little pocket listing suitable to put in your wallet (see me for example). It
assumes you have phone and address relations such as the ones I use in PDB, and it only prin
people in your database that (1) have phone numbers and/or addresses specified, and (2) are
connected by a member relship to a "phone list" entity (in case you have some you want to

Figure 7-3c. A message the user selected from the message set in Figure 7-3b.

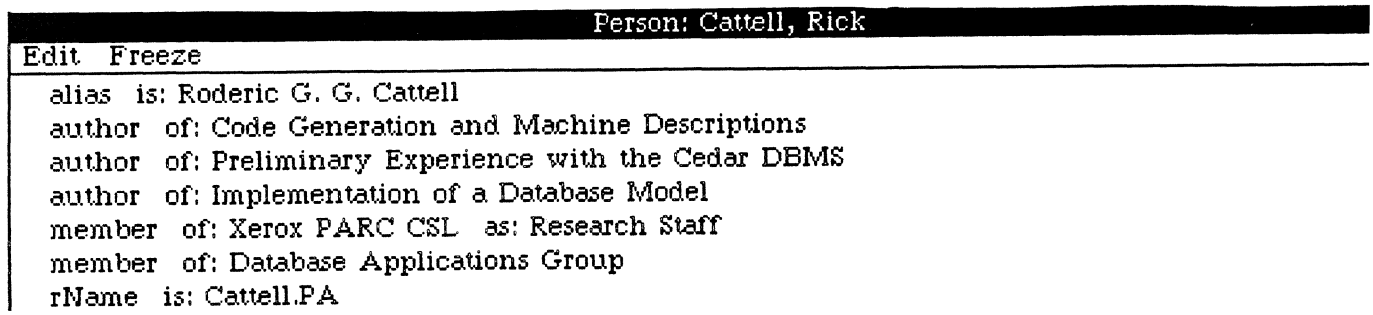

Person: Cattell, Rick
Edit Freeze
alias is: Roderic G. G. Cattell
author of: Code Generation and Machine Descriptions
author of: Preliminary Experience with the Cedar DBMS
author of: Implementation of a Database Model
member of: Xerox PARC CSL as: Research Staff
member of: Database Applications Group
rName is: Cattell.PA

Figure 7-3d. The person who sent the message, selected from the message in Figure 7-3c.

# 8. Status and conclusions

## 8.1 Summary

This document has described the Model level of the Cypress Database Management System, including the data model, the background and motivation for choosing that model, the client interface to the system, its implementation, and the database environment and applications for which it was implemented. The guiding principles in the design and implementation have been simplicity and utility for the set of applications we envision.

The result is a model that includes features of the relational model and distillations of desirable features of more recent semantic models. The model includes the concept of entities with unique names, a hierarchy of types of entities, types and uniqueness constraints on relation attributes, relational views, and a logical segmenting mechanism that can be used to facilitate physical distribution and independence of databases. This report discusses a number of issues in the implementation of these features, which are not present in any existing database system to the author's knowledge. The Cypress data model alleviates the problems motivating its development, reducing the quantity of data modelling mechanism built anew for each application, and simplifying the sharing of databases between applications. The Cypress model has also enabled the development of general-purpose tools formerly impractical due to the lack of type information and integrity checking in the Cedar database system.

## 8.2 Some results

Some overall statistics on the performance of the initial implementation may be helpful here. We developed two benchmark programs, one write-intensive and one read-intensive, to examine performance. Average times for the most common operations are roughly:

| | | |
|---|---|---|
| 1 | ms: | GetF |
| 1 | ms: | NameOf |
| 0.5 | ms: | NextEntity |
| 0.5 | ms: | NextRelship |
| 5 | ms: | DomainSubset |
| 7 | ms: | RelationSubset |
| 10 | ms: | SetF |
| 10 | ms: | ChangeName |

These times are approximately in order of decreasing frequency of calls by the benchmarks, and include overhead at all levels of the database and file systems. The times were taken on the Xerox Dorado, a personal super-computer with a micro-cycle time of about 60 ns. Note that since most of

the Cypress operations are disk-limited, the times increase only somewhat for slower processors. The times shown above vary widely with a particular application's data schema and access patterns, so these numbers should be regarded as very rough averages. Some effects of particular optimizations and schema changes were enumerated in Section 6.

A significant result of our work is that the type checking required by the data model is *not* a large overhead in the Cypress implementation. Because we do not compile database accesses, we must check in the implementation of every operation, e.g. SetF, that the arguments passed are of the proper and coordinated types. On a SetF, for example, we must check that: (1) the arguments are a relationship, attribute, and value, respectively; (2) the attribute is of the same relation as the relationship; (3) the value is of the same type as the attribute; and (4) that a key value constraint would not be violated by the new value. The caching of information about attributes improves the performance of the first three of these considerably. Without this caching, the GetF operation takes approximately 8 times as long.

A closer analysis of the time spent in a typical read operation, e.g. GetF, is enlightening. For our benchmark programs, the time breakdown was roughly as follows:

10% model level consistency checking and access path selection
20% storage level operation: actual read or update of data
50% waiting for disk operation (cache miss)
20% other overhead (page faults, garbage collection)

These figures suggest that work on co-location strategies might be fruitful, since a fair portion of the time is in disk operations. Note again, however, that statistics can vary widely with the particular application.

### 8.3 Status and plans

The first implementation of the Model level was completed in December of 1981, and was exercised and debugged through 1982. Approximately six man-months went into its development. This implementation includes essentially all data model features except views and augments. Views have been deferred to the development of the Query level.

Plans for the near future include development of more applications, continuing the work sketched in Section 7.

## Acknowledgments

Nori Suzuki and Mark Brown participated in the design and implementation of the original Cedar Database System. Peter Deutsch and Jerry Popek assisted in the design phase. Eric Bier assisted in the initial implementation of the Model level, as well as providing a useful sounding board for these ideas. Willie-Sue Haugeland co-implemented Walnut. Jim Donahue developed Hickory. Willie-Sue Haugeland, John Maxwell, and Jim Donahue have all helped with the Squirrel system. Mark Brown has maintained and elaborated the Cypress Storage level and was simultaneously the central designer and implementor of the Alpine file system which Cypress depends on for remote data storage.

Samuel Feldman, Dennis McLeod, Jim Donahue, Mark Brown, John Maxwell, Butler Lampson, William Kent, Ken Keller, Dushan Badal, and Peter Deutsch provided useful feedback on drafts of all or part of this document as it evolved over last year. Kathi Anderson and Subhana Menis helped prepare this report for publication.

## 9. Annotated bibliography

Abrial, J. R. Data Semantics, in Klimbie, J. and Koffman, K., *Database Management*, North-Holland, 1974.

Describes a data model based on binary relations. Relations that are N-ary even in functionally irreducible form must be represented by the introduction of artificial entities to represent relationships. See also Bracchi et al [1976] and Senko [1976] for binary relation models.

Armstrong, W. W. Dependency structures of data base relationships, *Proceedings IFIP Congress*, North-Holland, 1974.

This was one of the first studies of functional dependencies and relational normalization after Codd [1970]. LeDoux and Parker [1982] summarize more recent work.

Astrahan, M. M., et al. System R: Relational Approach to Database Management, *ACM Transactions on Database Systems*, **1**, 2, June 1976.

This is one of the most widely known and successful implementations of the relational data model. Much of the design of the original Cedar DBMS was derived from "RSS level" of this system.

Bachman, C. W. The Role Concept in Database Models, *Proceedings 3rd International Conference on Very Large Databases*, Tokyo, Japan, October 1977.

Concerned with behavioral properites of database objects, the treatment of entities in multiple roles, and integrating data models with programming languages. "Roles" for entities are actually more powerful than the sub-type hierarchy, although they are somewhat more complex.

Beeri, C., Bernstein, P. A., and Goodman, N. A Sophisticate's Introduction to Database Normalization Theory, *Proceedings 4th International Conference on Very Large Data Bases*, West Berlin, West Germany, 1978.

A discussion of relational normalization.

Biller, H. On the Notion of Irreducible Relations. *Database Architecture*, G. Bracchi and G. Nijssen, Eds. North-Holland, 1979.

Proposes a semantic definition of the irreducible relational form described in this paper. Contrasts the definition used in this paper with an earlier definition in terms of loss-less relational joins, showing our definition is more restrictive.

Bjorner, D. *Formalization of Data Base Models*. TR ID811, DCS, Technical University of Denmark, December 1978.

The relational, hierarchical, and network models are sequentially formally defined. Both the data definition and data modification/query are considered.

Bobrow, D. and Goldstein, I. Representing Design Alternatives, *Prooceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*, Amsterdam, July 1980. Also available as part of Technical Report CSL-81-3. Xerox Palo Alto Research Center, 1981.

A description of the *layer* concept in the PIE system. Layers are similar in concept to the augments discussed in this paper. However the basic data primitive is a Smalltalk object rather than relationships and entities, so the result does not provide the power of a database system, e.g., a query language. An implementation of layers was built, and feedback from that experience and the authors' applications built on it were invaluable in the implementation of augments.

Bobrow, D. An Overview of KRL, a Knowledge Representation Language, *Cognitive Science 1*, 1, 1977.

Describes a knowledge representation language KRL, intended for artificial intelligence applications. Many of the ideas in knowledge representation languages, most of which are present in KRL, are now being incorporated in database models (see Wong and Mylopoulos [1977]). With an implementation of these ideas such as Cypress, the result is a system with much of the representational flexibility of knowledge representation languages without the database system features they normally lack: performance, shared access, persistent storage, or large quantities of data.

Bracchi, G., Paolini, P., and Pelagatti, G. Binary Logical Associations in Data Modelling, *Proceedings IFIP TC-2 Working Conference on Modelling in Data Base Management Systems*, North-Holland, 1976.

Another discussion of binary relation models, similar to Abrial et al [1974]. Presents a number of arguments in favor of binary over N-ary relations, although these arguments are in reference to 3rd normal form rather than the irreducible form described in the present paper.

Brodie, M. L. Data Types and Databases, IFSM TR #37, University of Maryland, December 1978.

Relates data types and data models. We do not directly include this idea in Cypress, although relational views can be used to represent operations on object-oriented entity "data types," and the Squirrel interface is also oriented around the entity types. The paper references Brodie's thesis on data types and other topics.

Brodie, M. L. and Zilles, S. N., Eds. *Proceedings of Pingree Park Workshop on Data Abstraction, Databases, and Conceptual Modelling*, SIGMOD Record **11**, 2, February 1981.

A collection of papers relating data modelling from the points of view of programming languages, artificial intelligence, and databases. The same concepts (e.g., hierarchies of types) have independently been studied in these three areas. The Pingree Park workshop provided a useful interchange of ideas.

Brown, M., Cattell, R., and Suzuki, N. The Cedar Database Management System: A Preliminary Report, *Proceedings ACM SIGMOD Conference 1981*, Ann Arbor, April 1981.

An overview of the Cedar DBMS as it existed prior to the introduction of the Cypress data model, Squirrel tools, and Pilot and Alpine file systems.

Buneman, P. and Frankel, R. FQL: A Functional Query Language, *Proceedings of ACM SIGMOD International Conference on the Management of Data*, Boston, May 1979.

A "functional" data model and query language (see also Shipman[1981]).

Cattell, R., Donahue, J., Haugeland, W., and Maxwell, J. Integrating Database Applications in a Personal Computing Environment, submitted to *IFIP Conference on High-Performance Personal Computers*, July 1983.

This paper provides a more thorough discussion of the ideas behind integrating the database applications built on top of Cypress, using Squirrel.

Chan, A., Danberg, S., Fox, S., Lin, W., Nori, A., and Ries, D. Storage and Access Structures to Support a Semantic Data Model, *Proceedings International Conference on Very Large Data Bases*, 1982.

This is the only other implementation effort under way, of which the author is aware, to implement some of the features of recent semantic data models. The implementation is of an adaptation of DAPLEX to the programming language ADA (see Shipman [1981], Smith et al. [1981]). Although the implementation was still under way at the time of this writing, the design includes a number of the ideas used in our implementation, e.g., clustering of data on the basis of entities. They do not maintain back-pointers to assist in updating entity references when data is moved or the entity is destroyed, instead using a level of indirection in an "entity directory."

Chen, P. The Entity-Relationship Model–Towards a Unified View of Data, *ACM Transactions on Data Base Systems* **1**, 1, January 1976.

This was the earliest widely-referenced work on extensions to the relational model. It introduced entities and entity types (domains), and showed how they could be integrated with relationships, relations, and values.

Codd, E. F. A Relational Model of Data for Large Shared Data Banks, *CACM* **13**, 6, June 1970.

This was the earliest widely-referenced work defining the relational model. It introduced relationships, relations (relationship types), values, and value types.

Codd, E. F. Extending the Database Relational Model to Capture More Meaning, *ACM Transactions on Data Base Systems* **4**, 4, December 1979.

Codd assembles several proposed extensions to the relational model. The biggest contribution of this paper is in recognizing that the operations on data are at least as important a component of the data model as the structural data definition, a fact most of the literature has ignored. Codd suggests such operations.

Demers, A., Donahue, J., and Skinner, G. Data Types as Values: Polymorphism, Type-checking, Encapsulation, *Conference Record 5th ACM Symposium on Principles of Programming Languages*, 1978.

Proposes some more powerful mechanisms for types in programming languages. Russel, the authors' language, supports these mechanisms (but is not yet implemented).

Fagin, R. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems* **2**, 3, 1977.

Fagin, R. A Normal Form for Relational Databases That is Based on Domains and Keys. *ACM Transactions on Database Systems* **6**, 3, 1981.

The above two papers by Fagin describe two normal forms for databases, the second more restrictive than the first, and the first more restrictive than Codd's [1970].

Feldman, J. and Rovner, P. An Algol-based Associative Language, *CACM* **12**, 8 August 1969.

The LEAP language described here has many of the properties of more recent data models, and some interesting language-integrated ideas about *access* to the data (looping constructs and user-defined procedures for matching tuples). LEAP uses binary relations.

Hall, P., Oulett, J., and Todd, S. Relations and Entities, *Proceedings of the IFIP TC-2 Working Conference on Modelling in Data Base Management Systems*, 1976.

An earlier discussion of entities and the irreducible relations which Biller [1979] further formalizes.

Hammer, M. and McLeod, D. The Semantic Data Model: A Modelling Mechanism for Data Base Applications, *ACM Transactions on Database Systems 6*, 3 (September) 1981.

A more recent summary of the SDM data model described in McLeod's PhD disseration [1978].

Herot, C. The Spatial Data Management System, *Proceedings International Conference on Very Large Data Bases*, Rio de Janeiro, Brazil, 1981.

Describes a novel user interface to a database system allowing browsing in a spatial layout of entities.

Israel, J., Mitchell, J., and Sturgis, H. Separating Data from Function in a Distributed File System, Technical Report CSL-78-5, Xerox Palo Alto Research Center, 1978.

Describes Juniper, the transaction-based file system on which the Cedar DBMS was constructed.

Kapp, D. and Leben, J. *IMS Programming Techniques*, Van Nostrand, 1978.

Describes IBM's IMS, the most widely used commercial database system, and it's access language DL/I. IMS uses a form of hierarchical data model, and comes in a number of versions (IMS/VS, IMS360) with somewhat different features.

Katz, R. and Wong, E. Heterogenous Data Models–Part 1: Semantic Issues, ERL memo UCB/ERL m79/56, UC Berkeley, 1979.

Discusses issues in providing an integrated database system, composing database system components based on different data models. This is a hard and generally unsolved problem.

Kent, W. *Data and Reality*, North-Holland, 1978.

This book has an excellent thorough discussion of philosophical and epistemological aspects of entities, entity types, relations, and issues only briefly mentioned in most recent data model literature. There are little or no new ideas here, the purpose is to revisit questions inadequately studied elsewhere, e.g., "What is an entity?".

Kent, W. Limitations of Record-based Information Models, *ACM Transactions on Database Systems*, 4, 1 March 1979.

A good summary of limitations of the relational model, and some of the advantages of more recent data models.

Kent, W. Choices in Practical Data Design, *Proceedings 8th International Conference on Very Large Databases*, Mexico City, Mexico, September 1982.

Discusses some important unsolved issues in data modelling and schema design. One issue is whether an entity name composed of multiple parts or a relation with multiple attributes should be collapsed into a single name/attribute or kept separate (examples are month/day/year, or city/state/country). Another issue is whether to encode information as type or as data (for example, whether to include sex as a property of persons, or to declare different subtypes of Person for men and women). Other issues are "nameless" entities and uniqueness of entity names when there are several subtypes of a domain.

Kent, W. The Entity Join, *Proceedings 5th International Conference on Very Large Databases*, 1979.

Discusses the distinction between entities and their names.

Kerschberg, L., Klug, A., and Tsichritzis, D. A Taxonomy of Data Models, *Systems for Large Data Bases*, P. Lockemann and F. Neuhold, Eds., North-Holland, 1976.

This paper is a brief survey of 23 data models along some global dimensions: graph-theoretic versus set-theoretic, mathematical foundation, terminology, and level of abstraction.

King, R. and McLeod, D. Semantic Database Models, *Database Design*, S. B. Yao, Ed., 1982.

A recent survey of data models (see also Lochovsky and Tsichritzis [1982]).

LeDoux, C. and Parker, D. Reflections on Boyce-Codd Normal Form. *Proceedings 8th International Conference on Very Large Data Bases*, Mexico City, 1982.

A recent study of relational normalization.

Lindgreen, P. Basic Operations on Information as a Basis for Data Base Design, *Proceedings IFIP Congress*, North Holland, 1974.

A binary-relation data model.

McLeod, D. A Semantic Database Model and its Associated Structured User Interface, PhD Thesis, EE and CS, MIT, 1978.

Describes the SDM data model, including most of the ideas in the Cypress data model and some other ones, such as the introduction of *events*.

Mitchell, J., Maybury, W., and Sweet, R. Mesa Language Manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1979

Describes Mesa 5.0, the language in which the Cedar database system was originally implemented. No reference is externally available at the time of this writing for Cedar Mesa, the language in which the Cypress database system is implemented. The Cedar Mesa language of course strongly influenced the design of the programmer's interface to the Cypress data model.

Mylopoulos, J., Bernstein, P., and Wong, H. A Preliminary Specification of TAXIS: A Language for Designing Information Systems, TR CCA-78-02, Computer Corporation of America, Boston, January 1978.

Integrates a number of the concepts discussed in recent data models, including generalization hierarchies and specialization.

Pirotte, A. The Entity-Property-Association Model: An Information-Oriented Data Base Model, MBLE Report R343, March 1977.

Another early data model.

Quillian, M. R. Semantic Memory, in M. Minsky, Ed., *Semantic Information Processing*, MIT Press, 1968.

This is the earliest widely-referenced work with semantic nets, which have been used widely in Artificial Intelligence. Semantic nets, as they are generally used, allow entities and relationships but lacked some capabilities the relational model allows (e.g., queries on relations).

Redell, D, et al. Pilot: An Operating System for a Personal Computer, *Proceedings 7th Symposium on Operating Systems Principles*, Asilomar, (ACM order #534790) 1979.

Describes the Pilot operating system, on which the Cedar programming environment and Cedar DBMS depend. The Pilot file system provides a local transaction-based access to data for the Cedar DBMS (Juniper, described by Israel et al [1979], provides a remote transaction-based file system).

Rissanen, J. Independent Components of Relations, *ACM TODS* **2**, 4, December 1977.

A study of what Biller [1979] calls a-irreducible relational form, which is less restrictive than the irreducible form we use.

Roussopoulos, N. and Mylopoulos, J. Using Semantic Networks for Database Management, *Proceedings 2nd International Conference on Very Large Databases*, September 1975.

Discusses a data model based on semantic networks (Quillian [1968]).

Sadri, F. and Ullman, J. The Interaction Between Functional Dependencies and Template Dependencies. *Proceedings ACM SIGMOD Conference 1980*, Santa Monica.

Describes more complex kinds of dependencies and relational normal forms.

Schmidt, H. A. and Swenson, J. R. On the Semantics of the Relational Data Model, in King, F. Ed., *Proceedings ACM SIGMOD Conference*, San Jose, 1975.

An early paper discussing shortcomings of the relational model.

Senko, M. E. Data Structures and Data Accessing in Data Base Systems Past, Present, Future, IBM Systems Journal **16**, 3, 1977(a).

A binary-relation data model with separate physical, conceptual, and user levels.

Shipman, D. W. The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems* **6**, 1, March 1981.

Describes a data model, the functional data model, which is superficially quite different but actually very similar to the Cypress model. The Functional data model takes an attractively simple point of view, decomposing relationships into *functions* mapping values to values. Thus all the notation and terminology of mathematical functional notation is available for use. The "properties" described in this paper have some similarities to this idea, but Shipman makes this the basis for the model.

Shoshani, A. CABLE: A Language Based on the Entity-Relationship Model, Lawrence Berkeley Lab, Berkeley, California, 1978.

CABLE is a query language for Chen's Entity-Relationship model, based on chains of relationships between entities. For example, one might ask for all the people that are managers of departments that produce parts that are sold to Company X. Because the chains seem very natural for expression of queries, we are considering a similar kind of language for use in Cypress.

Smith, J.M. and Smith, D.C. Data Base Abstractions: Aggregation and Generalization, *ACM Transactions on Database Systems* 2, 2, June 1977(a).

A discussion of the idea of hierarchies of types.

Smith, J.M. and Smith, D.C. Integrated Specifications for Abstract Systems, TR UUCS-77-112, Computer Science, University of Utah, September 1977(b).

Concerned with combining structural and behavioral specifications.

Smith, J., Fox, S., and Landers, T. Reference Manual for ADAPLEX, Technical Report, Computer Corporation of America, January 1981.

Describes ADAPLEX, embedding the DAPLEX (Shipman[1981]) language in the Ada programming language.

Stonebraker, M., et al. The Design and Implementation of INGRES, *ACM Transaction on Database Systems* **2**, 3, September 1976.

INGRES is one of the two most widely known implementations of the relational data model (the other is System R, Astrahan et al [1976]).

Stonebraker, M. Hypothetical Databases as Views, *ACM Proceedings ACM SIGMOD 81*, Ann Arbor, April 1981.

Further work on INGRES referenced above has suggested ways to define integrity constraints and hypothetical databases through the use of views, although there may be problems with doing this as an "add-on". Hypothetical databases are described in this reference.

Sundgren, B. Conceptual Foundation of the Infological Approach to Data Bases, *Data Base Management*, J. Klimbie and K. Koffeman, Eds., North Holland, 1974.

Early paper describing the *infological* data model, which has not been fleshed out to the degree of most of the other models discussed. The basic primitives are objects (entities) and their properties.

Tsichritzis, D. C. LSL–A link and selector language, *Proceedings ACM SIGMOD 76*, 1976.

A query language for a network model database.

Tsichritzis, D. and Klug, A., Eds. The ANSI/X3/SPARC DBMS Framework. Report of the Study Group on Database Management Systems. *Information Systems* 3, 4, 1978.

A comprehensive data model in the Entity-Relationship family.

Tsichritzis, D. and Lochovsky, F. *Data Models*, Prentice-Hall, 1982.

This book is probably the most extensive and up-to-date discussion of data models currently available. It deals with conceptual models rather than physical models and complete systems, since many of the models described have not been implemented. The book covers the relational, network, hierarchical, entity-relationship, binary, semantic network, and infological data models.

Wiederhold, G. *Database Design*, McGraw-Hill, New York, 1977.

A good description of database systems from the point of view of implementation and physical access structures as opposed to the conceptual data model.

Wiederhold, G. and El-Masri, R. The Structural Model for Database Design, *Proceedings of the International Conference on the Entity-Relationship Approach to Systems Analysis and Design*, Los Angeles, December 1979.

Describes the "structural" data model, further described in El-Masri's PhD dissertation. The structural model classifies relations according to their semantic use, much as in Codd's [1980] extensions of the relational model.

Wong, H. and Mylopoulos, J. Two Views of Data Semantics: A Survey of Data Models in Artificial Intelligence and Database Management. *INFOR* **15**, 3, October 1977.

Contrasts knowledge representation languages and data models.

Zloof, M. M. Query by Example: The Invocation and Definition of Tables and Forms. *Proceedings First International Conference on Very Large Data Bases*, 1975.

The first widely-referenced work on querying databases by filling out forms, an idea used in Squirrel queryer windows.

## 10. Appendix: Axioms and formal definitions

The definition of a database includes a set of domains **D** (the domain domain), a set of relations **R** (the relation domain), a set of attributes **A** (the attribute domain), and a set of datatypes **T** (the datatype domain). We also define a subdomain function **sd** mapping **D** to **D**. The function **sd** must define a partial ordering of **D**.

We define the fundamental type constraint for a database:

> *The Attribute Type Constraint.* (a) Every relationship in a relation $R$ must have the same number of elements. We assign identifiers for corresponding elements of the relationships belonging to $R$, the attributes $a_1$, $a_2$, $a_3$, ... $a_N$. (b) For any of these attributes $a_i$, the value associated with $a_i$ for every relationship in $R$ must be an element of the same datatype or domain $d_i$, or of a domain $s_i$ in the transitive closure of **sd** applied to $d_i$.

Because every relationship in a relation $R$ has the same attributes, we may unambiguously refer to these as simply *R's attributes.* Every relationship in $R$ has a value of the same type associated with an attribute $A$, so we define a function **at** mapping **A** to the union of **D** and **T**, and call **at**($A$) $A$'s *type.*

*Relation keys*

Each data schema includes a uniqueness function **au** mapping **A** to the set {none, keypart}. If a relation R has M attributes, of which some subset $a_1$, $a_2$, ... $a_N$ (where N < M) have **au**($a_i$) = keypart, then no more than one relationship in R is permitted to have the same value associated with attributes $a_1$, $a_2$, ... $a_N$.

*Normalization*

For a relation R of order N, denote the predicate whose truth in the represented world corresponds to the presence of a relationship in the relation by $R^p[a_1, a_2, ... a_N]$. The arguments $a_i$ correspond to the attributes of R. We then define:

> A relation R is in *irreducible* form iff it cannot be split into relations $R_1$, $R_2$, ... $R_k$ such that for all possible states of the represented world:
>
> $$R^p[x_1, x_2, ... x_N] \leftrightarrow R_1^p[x_{i1}, ... x_{ik}] \wedge ... \wedge R_k^p[x_{j1}, ... x_{jl}]$$
>
> where
>
> $$\{x_{i1}, ... x_{ik}\} \subseteq \{x_1, x_2, ... x_N\}$$
>
> ...
>
> $$\{x_{j1}, ... x_{jl}\} \subseteq \{x_1, x_2, ... x_N\}$$

A relation is in *functionally irreducible form* iff (1) it is in irreducible form or (2) it is the natural relational join on a key attribute of two or more binary relations, and there is a dependency between the other (non-joined) attributes $a_1$, $a_2$, ... $a_N$ such that the semantics of attribute $a_i$ is dependent upon the value of $a_{i+1}$, for i = 1, 2, .. N-1.

*Axioms*

Below are the beginnings of an equational specification of the DB interface developed by Jim Donahue of the Computer Science Lab. Currently missing are complete descriptions of the schema operations (DeclareDomain, DeclareRelation, DeclareAttribute, . . .), some of the detailed description of the behavior of operations on erroneous arguments (the type-checking of RelationSubset, for instance), and any discussion of segments, transactions and initialization. A complete description will appear in a forthcoming paper.

In this style of specification, we present the semantics of the operations in terms of behaviorally equivalent programs composed of sequences of Cypress operations. Since the only way you can interrogate entities is through DomainSubset, the only important property of DeclareEntity is that it may affect the value of a later invocation of DomainSubset. The equations specify this by stating that certain sequences of Cedar statements are equivalent.

In the following, we will use the abbreviation

$$(s_1;\ .\ .\ .;\ s_n)\ \text{THEN}\ e_1 \leftrightharpoons e_2$$

where $s_1$, ...., $s_n$ are Cedar statements and $e_1$ and $e_2$ are Cedar expressions to mean that

$$\{\ s_1;\ ....;\ s_n;\ \text{RETURN}[e_1]\ \} \leftrightharpoons \{\ s_1;\ ....;\ s_n;\ \text{RETURN}[e_2]\ \}$$

(Quite often in what follows, the equivalences will be between Cedar procedure bodies; if Cedar were an expression language then the equivalences would be somewhat simpler to state.)

*Basic Operations on Entities*

The fundamental property of entities is that no two entities with the same name and same domain can exist (this is not true for relationships). The operations that manipulate entities include:

DeclareEntity -- add (or look up) an entity in a particular domain
DestroyEntity -- remove an entity from a domain
NameOf -- give the name of an entity

DomainOf -- give the domain of an entity
Eq -- compare two entities
Null -- has an entity been destroyed?
DomainSubset -- produce the subset of a domain with names in a given range
NextEntity -- sequence through an entity set
ReleaseEntitySet -- free the storage used by entity set (affects performance only)
DeclareEntity

*DeclareEntity*

The basic operation on entities is DeclareEntity, which either creates a new entity in a particular domain or returns a previously existing one. Its semantics is given in the following equations:

1. Declare produces an entity with the right name and domain:
   (e: Entity = DeclareEntity[d, name, NewOnly]) THEN
   [ DomainOf[e], NameOf[e] ] ⇋ [ d, name ]

2. NewOrOld is only a convenience:
   DeclareEntity[d, name, NewOrOld] ⇋
   { e: Entity = DeclareEntity[d, name, OldOnly];
   IF e = NIL THEN RETURN[DeclareEntity[d, name, NewOnly]] ELSE RETURN[e] }

3. Declare OldOnly is the same as domain subset:
   DeclareEntity[d, name, OldOnly] ⇋
   { es: EntitySet = DomainSubset[d, name];
   e: Entity = NextEntity[es]; ReleaseEntitySet[es]; RETURN[e] }

4. Declare NewOnly fails if the entity already exists:
   IF DeclareEntity[d, name, OldOnly] # NIL THEN DeclareEntity[d, name, NewOnly] ⇋
   IF DeclareEntity[d, name, OldOnly] # NIL THEN SIGNAL Error[AlreadyExists]

5. Declare doesn't work on system domains:
   DeclareEntity[DomainDomain, name] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareEntity[RelationDomain, name] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareEntity[AttributeDomain, name] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareEntity[DataTypeDomain, name] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareEntity[IndexDomain, name] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareEntity[IndexFactorDomain, name] ⇋ SIGNAL Error[ImplicitSchemaUpdate]

6. DeclareEntity doesn't work for NIL or null arguments:
   DeclareEntity[NIL, name] ⇋ SIGNAL Error[NILArgument]
   IF Null[d] THEN DeclareEntity[d, name] ⇋
   IF Null[d] THEN SIGNAL Error[NullifiedArgument]

*DestroyEntity*

Declare may put something in the database; Destroy removes it.

1. An entity that is destroyed will not be found:
   ( d: Domain = DomainOf[e]; name: ROPE = NameOf[e]; DestroyEntity[e] ) THEN

```
DeclareEntity[d, name, OldOnly] ⇋ NIL
```

2. All relships that reference an entity are destroyed when the entity is destroyed:
```
(SetF[rship, attr, e]; DestroyEntity[e]) THEN Null[rship] ⇋ TRUE
```

3. Once destroyed, an entity becomes Null:
```
(DestroyEntity[e]) THEN Null[e] ⇋ TRUE
(DestroyEntity[e1]) THEN Eq[e1, e2] ⇋ Null[e2]
```

4. Destroy doesn't work on NIL or null arguments (can't destroy the same entity twice):
```
DestroyEntity[NIL] ⇋ SIGNAL Error[NILArgument]
IF Null[e] THEN DestroyEntity[e] ⇋
IF Null[e] THEN SIGNAL Error[NullifiedArgument]
```

*DomainSubset*

DomainSubset returns an EntitySet that, if cycled through using NextEntity, contains all of the entities in the domain having names in the specified range, and only those entities.

1. DomainSubset finds only those entities having names between low and high:
```
( es: EntitySet = DomainSubset[d, low, high];
  e: Entity = DeclareEntity[d, name, NewOnly] ) THEN
DomainSubset[d, low, high] ⇋
IF name < low OR name > high THEN es ELSE DomainSubset[d, low, high]
```

2. DomainSubset finds all those entities having names between low and high:
```
( e: Entity = DeclareEntity[d, name, NewOnly] ) THEN
name >= low AND name <= high ⇋
{ es: EntitySet = DomainSubset[d, low, high];
  FOR next: Entity ← NextEntity[es], NextEntity[es] UNTIL next = NIL DO
    IF Eq[next, e] THEN {ReleaseEntitySet[es]; RETURN[TRUE]}
  ENDLOOP;
  ReleaseEntitySet[es]; RETURN[FALSE] }
```

*Equality of entities*

Two entities are equal iff they are both null or their names are the same and they are in the same domain.

```
Eq[e1, e2] ⇋
{ IF Null[e1] OR Null[e2] THEN RETURN[ Null[e1] AND Null[e2] ];
  RETURN[ Rope.Equal[NameOf[e1], NameOf[e2]] AND
    Eq[DomainOf[e1], DomainOf[e2]]] }
```

*Basic Operations on Relationships*

Relationships are tuples; the basic operations on relationships include setting and fetching the values of fields and searching the database for all tuples whose fields have values in specified ranges. The operations that manipulate relationships include:

DeclareRelship -- add (or look up) a relationship in a particular relation
DestroyRelship -- remove a relationship from a relation
RelationOf -- give the relation of a relationship
Eq -- compare two relationships
Null -- has a relationship been destroyed?
RelationSubset -- produce the subset of a relation with fields in a given range of values
NextRelship -- sequence through a relationship set
ReleaseEntitySet -- free storage used by a relationship set (affects performance only)

Unlike entities, it is possible to have the several tuples in the database with the same set of field values. (One can protect against this by declaring some of the attributes for a relation to be keys. Then checking is performed when declaring new relationships or setting the fields of existing relationships to guarantee that no other tuple exists with the same value for a key attribute. The equations given below ignore this additional complexity.) Allowing multiple tuples with the same values appears in the lack of an equation of the form "DeclareRelship followed by DeclareRelship with the same arguments and version OldOnly fails."

*DeclareRelship*

Most of these equations closely parallel the specification of DeclareEntity.

1. Declare produces a relationship with the proper values for each specified field:
   ( init: AttributeValueList = LIST[ [a1, v1], ..., [an, vn] ];
   rs: Entity = DeclareRelship[r, init] ) THEN GetF[ rs, ai ] ⇋ vi
2. Declare NewOrOld is a convenience:
   DeclareRelship[r, list, NewOrOld] ⇋
   { rs: DeclareRelship[r, list, OldOnly];
   IF rs = NIL THEN RETURN[DeclareRelship[r, list, NewOnly]] ELSE RETURN[rs] }
3. Again, Declare OldOnly is the same as the subset operation ___ this time, though, it may fail because of a multiple match:
   DeclareRelship[r, list, OldOnly] ⇋
   { rset: RelshipSet = RelationSubset[r, list];
   rs: Relship = NextRelship[rs];
   IF NextRelship[rset] = NIL THEN { ReleaseRelshipSet[rs]; RETURN[rs] }
   ELSE { ReleaseRelshipSet[rset]; SIGNAL Error[MultipleMatch] } }
4. Declare doesn't work on system domains:
   DeclareRelship[aRelation, list] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareRelship[aType, list] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareRelship[aUniqueness, list] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareRelship[aLength, list] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareRelship[aLink, list] ⇋ SIGNAL Error[ImplicitSchemaUpdate]
   DeclareRelship[dSubType, list] ⇋ SIGNAL Error[ImplicitSchemaUpdate]

5. DeclareEntity doesn't work for NIL or null arguments:
   DeclareRelship[NIL, list] ⇋ SIGNAL Error[NILArgument]
   IF Null[r] THEN DeclareRelship[r, list] ⇋
   IF Null[r] THEN SIGNAL Error[NullifiedArgument]

*DestroyEntity*

Because duplicates are allowed, the effect of Destroy is somewhat different. Declaring a relship NewOnly and destroying it has no effect if the relation is in the schema (note that the initialization of the fields of the relship is described below as applications of SetF).

1. Destroying a newly created relationship has no effect:
   { rs: Relship = DeclareRelship[r, [], NewOnly]; DestroyRelship[rs] } ⇋
   [] ← DeclareRelation[ NameOf[r], SegmentOf[r], OldOnly ]
2. If an existing relationship is destroyed, then it will not be found:
   ( rs: Relship = DeclareRelship[r, list, OldOnly]; DestroyRelship[rs] ) THEN
   DeclareRelship[r, list, OldOnly] ⇋ NIL
3. Once destroyed, an relationship becomes Null:
   (DestroyRelship[rs]) THEN Null[rs] ⇋ TRUE
   (DestroyRelship[rs1]) THEN Eq[rs1, rs2] ⇋ Null[rs2]
4. Destroy doesn't work on NIL or null arguments (can't destroy the same entity twice):
   DestroyRelship[NIL] ⇋ SIGNAL Error[NILArgument]
   IF Null[rs] THEN DestroyRelship[rs] ⇋
   IF Null[rs] THEN SIGNAL Error[NullifiedArgument]

*RelationSubset*

RelationSubset returns a RelshipSet that, if cycled through using NextRelship, contains all of the relationships in the relation having field values in the specified ranges.

1. RelationSubset finds only relships satisfying the AttributeValueList:
   ( init: AttributeValueList = LIST[ [a1, v1], . . ., [an, vn] ]
   search: AttributeValueList = LIST[ [a1, l1, h1], . . ., [an, ln, hn] ];
   rset: RelshipSet = RelationSubset[ r, search ];
   [] ← DeclareRelship[r, init, NewOnly ] ) THEN
   RelationSubset[ r, search ] ⇋
   IF v1 < l1 OR v1 > h1 OR . . . vm < lm OR vm > hm
   THEN rset ELSE RelationSubset[ r, search ]
2. And RelationSubset finds all relships satisfying the AttributeValueList:
   ( init: AttributeValueList = LIST[ [a1, v1], . . ., [an, vn] ]
   search: AttributeValueList = LIST[ [a1, l1, h1], . . ., [an, ln, hn] ];
   rs: DeclareRelship[r, init, NewOnly];
   rset: RelshipSet = RelationSubset[ r, search ] ) THEN
   v1 >= l1 AND v1 <= h1 AND . . . vm >= lm AND vm <= hm ⇋
   { FOR next: Relship ← NextRelship[rs], NextRelship[rs] UNTIL next = NIL DO
   IF Eq[ next, rs ] THEN { ReleaseRelshipSet[rs]; RETURN[TRUE] }
   ENDLOOP;
   ReleaseRelshipSet[rs]; RETURN[FALSE] }

*SetF and GetF*

SetF changes the fields of a tuple, GetF reads them. Here are also the axioms that describe the type-checking performed on assignment.

1. Setting the field of one relship changes its value for all equal relships (Eq for relships is essentially alias):
```
{ SetF[rship1, attr, val]; RETURN[ GetF[rship2, attr] ] } ⇋
{ oldVal: Value = GetF[rship2, attr];
    SetF[rship1, attr, val];
    RETURN[ IF Eq[rship1, rship2 ] THEN val ELSE oldVal ] }
```
2. Setting a field with its current value is a no-op (except for errors):
```
SetF[ rship, attr, GetF[rship, attr] ] ⇋
{ IF Eq[RelationOf[rship], aRelation] THEN SIGNAL Error[ImplicitSchemaUpdate];
  IF Eq[RelationOf[rship], aType] THEN SIGNAL Error[ImplicitSchemaUpdate];
  IF Eq[RelationOf[rship], aLength] THEN SIGNAL Error[ImplicitSchemaUpdate];
  IF Eq[RelationOf[rship], aUniqueness] THEN SIGNAL Error[ImplicitSchemaUpdate];
  IF Eq[RelationOf[rship], aLink] THEN SIGNAL Error[ImplicitSchemaUpdate];
  IF Eq[RelationOf[rship], ifIndex] THEN SIGNAL Error[ImplicitSchemaUpdate];
  IF Eq[RelationOf[rship], ifAttribute] THEN SIGNAL Error[ImplicitSchemaUpdate];
  [] ← GetF[rship, attr] }
```
3. Declare NewOnly can be defined in terms of setting the fields one at a time:
```
DeclareRelship[ r, LIST[ [a1, v1], . . ., [an, vn] ], NewOnly ] ⇋
{ rship: Relship = DeclareRelship[ r, [], NewOnly ];
    SetF[ rship, a1, v1 ]; . . .; SetF[ rship, an, vn ]; RETURN[rship] }
```
4. GetF fails if the attribute belongs to the wrong relation:
```
[] ← GetF[rship, attr] ⇋
{ IF rship = NIL OR attr = NIL THEN SIGNAL Error[NILArgument];
  IF Null[rship] OR Null[attr] THEN SIGNAL Error[NullifiedArgument];
  { rel1: Relation = RelationOf[rship];
      rel2: Relation = V2E[ GetP[ e: attr, aIs: aRelationIs, aOf: aRelationOf ] ];
      IF NOT Eq[rel1, rel2] THEN SIGNAL Error[ IllegalAttribute ] } }
```

The type-checking that goes on when a SetF is performed is fairly straightforward: If the value to be assigned is from one of the primitive types, then the type of the attribute (found in the aType relation) must be identical. In the case of assigning entity values, the domain of the value must be a subtype of the domain that is the type of the attribute. We define this by a set of similar equations for each of the primitive types and one for the entity-valued case.

5. Integer type-checking:
```
{ rship: Relship = DeclareRelship[r, [], NewOnly];
    SetF[ rship, attr, I2V[integer] ]; DestroyRelship[rship] } ⇋
{ [] ← GetF[rship, attr];
  { type: Entity = V2E[ GetP[ e: attr, aIs: aTypeIs, aOf: aTypeOf ] ];
    IF NOT Eq[type, IntType] THEN SIGNAL Error[MismatchedAttributeValueType] }}
```

6. Boolean type-checking:

```
{ rship: Relship = DeclareRelship[r, [], NewOnly];
    SetF[ rship, attr, B2V[boolean] ]; DestroyRelship[rship] } ⇋
{ [] ← GetF[rship, attr];
    { type: Entity = V2E[ GetP[ e: attr, aIs: aTypeIs, aOf: aTypeOf ] ];
     IF NOT Eq[type, BoolType] THEN SIGNAL Error[MismatchedAttributeValueType] }}
```

7. String type-checking:

```
{ rship: Relship = DeclareRelship[r, [], NewOnly];
    SetF[ rship, attr, S2V[string] ]; DestroyRelship[rship] } ⇋
{ [] ← GetF[rship, attr];
    { type: Entity = V2E[ GetP[ e: attr, aIs: aTypeIs, aOf: aTypeOf ] ];
    IF NOT Eq[type, StringType] THEN SIGNAL Error[MismatchedAttributeValueType] }}
```

8. Time type-checking:

```
{ rship: Relship = DeclareRelship[r, [], NewOnly];
    SetF[ rship, attr, T2V[time] ]; DestroyRelship[rship] } ⇋
{ [] ← GetF[rship, attr];
    { type: Entity = V2E[ GetP[ e: attr, aIs: aTypeIs, aOf: aTypeOf ] ];
    IF NOT Eq[type, TimeType] THEN SIGNAL Error[MismatchedAttributeValueType] }}
```

9. Entity type-checking (AnyDomainType is a wild card):

```
{ rship: Relship = DeclareRelship[r, [], NewOnly];
    SetF[ rship, attr, E2V[entity] ]; DestroyRelship[rship] } ⇋
{ [] ← GetF[rship, attr];
    { type: Entity = V2E[ GetP[ e: attr, aIs: aTypeIs, aOf: aTypeOf ] ];
      IF NOT (Eq[type, AnyDomainType] OR SubType[DomainOf[entity], type])
      THEN SIGNAL Error[MismatchedAttributeValueType] }}
```

*RelationOf*

RelationOf simply gives the relation of a relationship.

```
RelationOf[ DeclareRelship[ r, list ], NewOrOld ] ⇋
{ [] ← DeclareRelship[ r, list, NewOrOld ]; RETURN[r] }
```

*Operations on the database schema*

The database schema is stored in a set of system domains and relations. The operations on these objects include:

DeclareDomain -- add (or look up) a domain
DeclareRelation -- add (or look up) a relation
DeclareAttribute -- add (or look up) a field name of a relation
DestroyDomain -- destroy a domain (and all of its entities)
DestroyRelation -- destroy a relation (and all of its relationships)
DeclareSubType -- define one domain to be a subtype of another
DestroySubType -- remove the two domains from the dSubType relation

These operations behave much like DeclareEntity, DeclareRelship, DestroyEntity and DestroyRelship; it's just that they work on the system domains and relations.

*DeclareDomain/DestroyDomain*

1. DeclareDomain adds a new (empty) domain:
   ( d: Domain = DeclareDomain[ name, segment, NewOnly ] ) THEN
   { es: EntitySet = DomainSubset[d, NIL];
     e: Entity = NextEntity[es];
     ReleaseEntitySet[es]; RETURN[s] } ⇋ NIL

2. Again, NewOrOld is a convenience:
   DeclareDomain[name, segment, NewOrOld] ⇋
   { d: Domain = DeclareDomain[name, segment, OldOnly];
     IF d = NIL THEN RETURN[DeclareDomain[name, segment, NewOnly]]
     ELSE RETURN[d] }

3. Declare OldOnly is the same as domain subset on DomainDomain:
   DeclareDomain[name, segment, OldOnly] ⇋
   { es: EntitySet = DomainSubset[d: DomainDomain,
         lowName: name, searchSegment: segment];
     e: Entity = NextEntity[es]; ReleaseEntitySet[es]; RETURN[e] }

4. Declare NewOnly fails if the entity already exists:
   ([] ← DeclareDomain[name, segment, NewOrOld]) THEN
   DeclareDomain[name, segment, NewOnly] ⇋ SIGNAL Error[AlreadyExists]

5. Declare NewOnly changes DomainDomain:
   (d: Domain = DeclareDomain[name, segment, NewOnly]) THEN
   DeclareDomain[name, segment, OldOnly] ⇋ d

6. A domain that is destroyed will not be found:
   (name: ROPE = NameOf[d]; seg: Segment = SegmentOf[d]; DestroyDomain[d]) THEN
   DeclareDomain[name, segment, OldOnly] ⇋ NIL

7. Destroying a domain causes all of the entities in the domain to be destroyed:
   ( es: EntitySet = DomainSubset[d, NIL]; DestroyDomain[d] ) THEN
   { e: Entity = NextEntity[es]; ReleaseEntitySet[es]; RETURN[e] } ⇋ NIL

8. Destroying a domain causes entities to become Null:
   ( e: Entity = DeclareEntity[d, name]; DestroyDomain[d] ) THEN
   Null[e] ⇋ TRUE

9. Destroying a domain causes relationships that reference it to be destroyed:
   ( SetF[rship, attr, e]; DestroyDomain[d] ) THEN Null[rship] ⇋ TRUE

10. Destroy doesn't work on NIL or null arguments (can't destroy the domain twice):
    DestroyDomain[NIL] ⇋ SIGNAL Error[NILArgument]
    IF Null[d] THEN DestroyDomain[d] ⇋
    IF Null[d] THEN SIGNAL Error[NullifiedArgument]

*DeclareRelation/DestroyRelation*

1. DeclareRelation adds a new (empty) relation:
   ( r: Relation = DeclareRelation[ name, segment, NewOnly ] ) THEN
   { rs: RelshipSet = RelationSubset[r, NIL]; rship: Relation = NextRelship[rs];
   ReleaseRelshipSet[rs]; RETURN[rship] } ⇋ NIL

2. Again, NewOrOld is a convenience:
   DeclareRelation[name, segment, NewOrOld] ⇋
   { r: Relation = DeclareRelation[name, segment, OldOnly];
   IF r = NIL THEN RETURN[DeclareRelation[name, segment, NewOnly]]
   ELSE RETURN[r] }

3. Declare OldOnly is the same as domain subset on RelationDomain:
   DeclareRelation[name, segment, OldOnly] ⇋
   { es: EntitySet = DomainSubset[d: RelationDomain,
   lowName: name, searchSegment: segment];
   e: Entity = NextEntity[es]; ReleaseEntitySet[es]; RETURN[e] }

4. Declare NewOnly fails if the entity already exists:
   ([] ← DeclareRelation[name, segment, NewOrOld]) THEN
   DeclareRelation[name, segment, NewOnly] ⇋ SIGNAL Error[AlreadyExists]

5. Declare NewOnly changes RelationDomain:
   (r: Relation = DeclareRelation[name, segment, NewOnly]) THEN
   DeclareRelation[name, segment, OldOnly] ⇋ r

6. A relation that is destroyed will not be found:
   (name: ROPE = NameOf[r]; seg: Segment = SegmentOf[r]; DestroyDomain[r]) THEN
   DeclareRelation[name, segment, OldOnly] ⇋ NIL

7. Destroying a domain causes all of the relships in the domain to be destroyed:
   ( rs: RelshipSet = RelationSubset[r, NIL]; DestroyRelation[r] ) THEN
   { rship: Relation = NextRelship[rs]; ReleaseRelshipSet[rs]; RETURN[rship] } ⇋ NIL

8. Destroying a domain causes relships to become Null:
   ( rship: Relation = DeclareRelship[r, [], NewOnly]; DestroyRelation[r] ) THEN
   Null[rship] ⇋ TRUE

9. Destroy doesn't work on NIL or null arguments (can't destroy the relation twice):
   DestroyRelation[NIL] ⇋ SIGNAL Error[NILArgument]
   IF Null[r] THEN DestroyRelation[r] ⇋ IF Null[r] THEN SIGNAL Error[NullifiedArgument]

*DeclareAttribute*

The important properties of DeclareAttribute are the changes made to the system relations aType, aLength, aUniqueness and aLink when a new attribute is declared. If a type, length, uniqueness or link is specified when the attribute is looked up (Declare with version OldOnly), then these additional arguments are checked for consistency with the values stored in the relations. Note that there is no corresponding DestroyAttribute.

1. DeclareAttribute adds a new attribute:
   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] )
   THEN DeclareAttribute[r: r, name: name, version: OldOnly] ⇋ a

2. DeclareAttribute changes the aType relation:
   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] ) THEN
   V2E[ GetP[ e: a, aIs: aTypeIs, aOf: aTypeOf ] ] ⇋ type

3. DeclareAttribute changes the aUniqueness relation:
   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] )
   THEN V2I[ GetP[ e: a, aIs: aUniquenessIs, aOf: aUniquenessOf ] ] ⇋
   LOOPHOLE[uniqueness]

4. DeclareAttribute changes the aLength relation:
   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] )
   THEN V2I[ GetP[ e: a, aIs: aLengthIs, aOf: aLengthOf ] ] ⇋ length

5. DeclareAttribute changes the aLink relation:
   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] )
   THEN V2B[ GetP[ e: a, aIs: aLinkIs, aOf: aLinkOf ] ] ⇋ link

6. Again, NewOrOld is a convenience:
   DeclareAttribute[r, name, type, uniqueness, length, link, NewOrOld] ⇋
   { a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, OldOnly];
   IF a = NIL
   THEN RETURN[DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly]]
   ELSE RETURN[a] }

7. Declare OldOnly is the same as domain subset on AttributeDomain:
   DeclareAttribute[r: r, name: name, version: OldOnly] ⇋
   { es: EntitySet = DomainSubset[d: AttributeDomain,
   lowName: name, searchSegment: SegmentOf[r]];
   e: Entity = NextEntity[es]; ReleaseEntitySet[es]; RETURN[e] }

8. Declare OldOnly checks any supplied type, length, link, uniqueness:
   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] )
   THEN DeclareAttribute[r: r, name: name, type: type', version: OldOnly] ⇋
   IF NOT Eq[type', type] THEN SIGNAL Error[MismatchedExistingAttribute]
   ELSE DeclareAttribute[r: r, name: name, version: OldOnly]

   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] )
   THEN DeclareAttribute[r: r, name: name, uniqueness: uniqueness', version: OldOnly]
   ⇋ IF NOT uniqueness = uniqueness' THEN
   SIGNAL Error[MismatchedExistingAttribute]
   ELSE DeclareAttribute[r: r, name: name, version: OldOnly]

   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] )
   THEN DeclareAttribute[r: r, name: name, length: length', version: OldOnly] ⇋
   IF NOT length = length' THEN SIGNAL Error[MismatchedExistingAttribute]
   ELSE DeclareAttribute[r: r, name: name, version: OldOnly]

   ( a: Attribute = DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] )

THEN DeclareAttribute[r: r, name: name, link: link', version: OldOnly] ⇋
IF NOT link = link' THEN SIGNAL Error[MismatchedExistingAttribute]
ELSE DeclareAttribute[r: r, name: name, version: OldOnly]

9. Declare NewOnly fails if the entity already exists:
([] ← DeclareAttribute[r, name, type, uniqueness, length, link, NewOrOld]) THEN
DeclareAttribute[r, name, type, uniqueness, length, link, NewOnly] ⇋
SIGNAL Error[AlreadyExists]

10. Declare doesn't work on NIL or null arguments:
DeclareAttribute[NIL, name, type, uniqueness, length, link] ⇋
SIGNAL Error[NILArgument]
IF Null[r] THEN DeclareAttribute[r, name, type, uniqueness, length, link] ⇋
IF Null[r] THEN SIGNAL Error[NILArgument]

## Index of common terms

attribute: column of relation 6, 34, 57, 59
augments: mechanism for encapsulating updates 21, 69
basic operations: DeclareEntity, GetF, etc. 10, 68, 81
caching data schema: used in implementation 77
changing entity names 16, 39
conceptual data model 5
data independence 5
data model: type mechanism DBMS provides 1, 5
data schema: types in a particular database 1, 11
datatype: type of datum 6, 27
datum: string, integer, etc. 5, 27
domain: type of entity 6
entity: database representative of an object 5, 27, 57, 75
entity/relationship distinction 59, 60
entity/datum distinction 5, 57
errors: generated by illegal operations 45
external entity: real world object 8
functionally ireducible: normal form 18, 65
indices, B-tree: used in implementation 35, 48
internal entity: we use "entity" to mean this 9
irreducible: normal form 18, 65
key: unique attributes of a relation 9, 34, 64
links and colocation: connecting entities & relationships 35, 78
lists and sets: representation in our model 49, 62
name: of an entity 9, 63
normalization: of relations 16, 51, 65
physical data model 5, 74
properties: simulated "attributes" of entities 14, 9, 36, 39, 59
query operations: DomainSubset, RelationSubset 11, 39, 68, 83
referential integrity: reason for entities 57
relation: type of relationship 6, 34, 57
relational data model 1, 9
relationship: a tuple of entity and datum values 6, 7, 27, 60, 75
schema declaration operations: DeclareDomain, etc. 12, 32
segments: mechanism for separating databases 18, 69
simplicity and utility: criteria for data model choice 4
surrogate relation optimization 80
system entities, domains, relations 11, 43
transaction and segment operations 29
translucent attributes: accessing attributes as strings 15, 38
type hierarchy: domains and subdomains 6, 28, 66
undefined values: attributes never assigned 38
update anomaly: reason for normalization 17
value: an entity or datum 6, 76
views: relations that are not stored 24, 69